44100113-0: COMPUTER NETWORKS
FALL 2024

Socket Programming

Assigned: September 25, 2024
Due Date: 23:59, October 20, 2024

Section 1-2 describe the assignment.

Section 3-6 provide very useful information to finish this project.

1 ASSIGNMENT

You must work on this assignment individually.

This project is designed for you to:

- Learn to setup communications between PCs with socket.
- To be familiar with FTP protocol.
- To be familiar with network protocol stack.
- Learn to build client/server applications that communicate using sockets.

Before writing this project, you need to finish the following task, which gives you basic ideas for UDP programming:

- Read the given programs located in `udp` directory carefully. You may choose any language that you are familiar with.
- Modify the UDPServer. The server counts the number of messages it received and records it as a sequence number. For each string the server received, add the sequence number before the received string and return the whole string back to the client. For example, if the server received "hello" and the sequence number now is 1, return the string "1 hello" back to the client.
- Modify the UDPClient. Let the client send messages with the number from 0 to 50 to the server. The server then returns all messages with the sequence numbers described above.
- Answer: How to write a chat program (two **clients** chat with each other) with UDP?

Then, you will be asked to create, from scratch, a miniature **FTP server** with the following characteristics:

- Your server must serve files from a designated directory on your system to clients making requests on a designated TCP port.

- Your server must handle USER, PASS, RETR, STOR, QUIT, SYST, TYPE, PORT, PASV, MKD, CWD, PWD, LIST, RMD commands from the clients. You may support other commands.
- You may assume that all data connections will be binary - i.e., you can choose to ignore TYPE A requests. Binary mode means that you should transfer the data without modifying/converting it in any way.
- You must use the Berkeley Socket API and write your server in **C**, which indicates that you must write your server running with GNU/Linux. You may not use any libraries containing code specifically designed to implement FTP functionality. We suggest that you compile your code with gcc or clang.
- Each Unix vendor has its own special version of the "make" program builder, and each one has its own special beloved features. For this project, we will require that you use GNU Make. The makefile for this project should be trivial.
- If your server emits debugging or trace information on the standard output or standard error streams, it should suppress this output.
- Handling invalid input reasonably and generating defensible error codes.
- Support integral large file transmission.
- Support connections from multiple clients simultaneously.
- (Optional) Resume transmission after connection terminated.
- (Optional) File transmission without blocking the server.

Finally, you will also be asked to create a simple **FTP client** with the following characteristics:

- Your client must run with GNU/Linux. You can write your client in your favorite programming language, but any FTP library is not allowed. Besides, you are required to provide both source code and executable file of your client in your submission.
- Your client must use USER, PASS, RETR, STOR, QUIT, SYST, TYPE, PORT, PASV, MKD, CWD, PWD, LIST, RMD commands to log in a common server and download/upload files as well as manipulating directories.
- You may simply use binary mode.
- Your client is able to log in a provided commercial FTP server and download/upload files.
- Support integral large file transmission.
- (Optional) Resume transmission after connection terminated.
- (Optional) User-friendly GUI.
- (Optional) File transmission without blocking the GUI.

NOTE: You will get 5 extra points for each optional completion, but **no more than** 10 extra points in total.

Project deliverables:

Your modified UDP programs must be placed in `udp` directory. Leave the filenames without any change. For the question in UDP part, write a plain text file in English or Chinese.

Your server must be named "server". It must accept, in any order, the following command line arguments:

- -port n: An ASCII string representing the TCP port number your server will bind to and listen for requests on. If you do not receive this argument, default to port 21.
- -root /path/to/file/area: The pathname of the directory tree that will serve as the root for all requests. If you do not receive this argument, default to "/tmp".

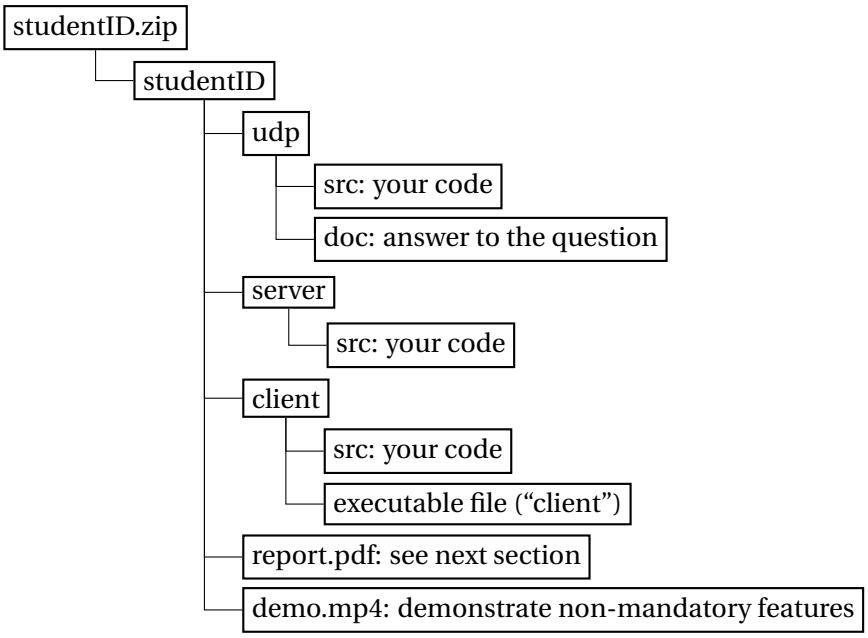
Your client must be named "client". It must accept, in any order, the following command line arguments:

- -ip IPaddress: The IP address of the FTP server to connect to, in the format "xxx.xxx.xxx.xxx". If you do not receive this argument, default to "127.0.0.1".
- -port n: An ASCII string representing the TCP port number that the FTP server is listening for re-

quests on. If you do not receive this argument, default to port 21.

If you complete any optional features, even those not mentioned in this document, please include a demonstration video with your submission.

Your files are to be organised as follows:



2 Grading Guidelines

- If your project generates compiler warnings, you will lose credit; if your server dumps core during our testing, you will lose substantial credit. Handling invalid input reasonably and generating defensible error codes are fundamental parts of writing any server program.
- Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the TA to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately. Putting all of your code in one module counts as an egregious design failure.
- Your project will be graded based on the following four parts:

  1. UDP programming (10)
     If you implement the program correctly, you will get full mark of 10.
  2. Implementation of FTP protocol (70 + 10)
     If you implement the server and it passes all test cases, you will get 40 credits. If you implement the client and it passes all test cases, you will get another 30 credits. If you complete any two optional requirements, you will get 10 extra credits.
     Please note that we have provided several test cases, i.e., `autograde_server.py` and `autograde_client.py`, in the issued zip file.
  3. Project report (20)
     You can include a short introduction to your FTP, FTP commands you implemented, etc. in your report.
     Anything you think valuable, e.g., difficulties you encountered, your innovative ideas, can be included in the report. However, the report should not be longer than THREE pages.

3 FTP BACKGROUND

An FTP server is a program that listens for incoming TCP connections (typically, but not always, on port 21). The client connects to the server on port 21. It sends commands to the server via this connection and receives the replies from the server over this same connection. These commands and replies are made up of special 3 digit codes and plain text strings. Actual data transfers (file transfers, directory listings etc.) is done via another connection specified via either the PORT or the PASV command. The PORT command allows a client to tell the server which IP address and port number to connect to to send/retrieve data. The PASV command allows a server to tell a client which IP address and port number to connect to to retrieve/send data. The FTP protocol has many optional features that allow an implementation to be bewilderingly complicated and bug-ridden.

Tutorial material on FTP is available in Ftp by Example (`https://www.cs.colostate.edu/helpdocs/ftp.html`); we suggest you start by reading this. D. J. Bernstein has a lot of information on the exact strings and commands expected by FTP servers and clients at `http://cr.yp.to/ftp.html`. As is the case with most Internet protocols, the authoritative definition is found in "Request for Comments" (RFC) documents. FTP was defined by Internet RFC 959 (`http://www.ietf.org/rfc/rfc959.txt`). This RFC defines everything about the basic FTP protocol. However, it is a long, dense document and we do not expect you to read it thoroughly. It is provided for your reference only. There have been additions to the FTP protocol in RFC 2640 (`http://www.ietf.org/rfc/rfc2640.txt`) and RFC 2228 (`http://www.ietf.org/rfc/rfc2228.txt`). You will not need to implement any of the additional features added to FTP by these 2 RFCs.

3.1 LOGGING IN (USER/PASS COMMANDS)

FTP servers require clients to log in before allowing those clients to access the files on the server. For the purposes of this assignment, you only need to implement anonymous logins. I.e., the username of the client should be anonymous. The server should respond with the appropriate string (which has the right command codes embedded in it) asking the client for the password, which should be just the users email address. The user should then be able to respond with an email address as the password. The strings to support are thus

1. When the client initially connects to the server, the server should then respond with an initial message. Something like "220 Anonymous FTP server ready.\r\n"
2. The server should then expect the client to send a "USER anonymous" command. All other commands are invalid. All users other than anonymous are not supported.
3. The server should then reply with a confirmation that the user anonymous is accepted and ask for an email address as a password.
4. The client should then respond with a "PASS some_password" string.
5. The server should then log the client in and reply with the greeting message.

However, you are encouraged to implement a user table to authenticate users.

Please refer to Section 4 for a complete example and to Section 3.5 for a description of the return codes the server is supposed to use.

3.2 PORT/PASV MODE

FTP works by using 2 connections. The 1st connection is made when the client connects to the servers listening port (usually port 21). All control traffic (commands from the client and responses from the server) are transferred over this connection.

However, FTP uses another connection to actually transfer file/directory information to the client. This can be the output of a LIST command or the contents of a file itself. I.e., one of these methods must be used before the client can execute a RETR, STOR or LIST command. FTP specifies 2 methods to do this.

PORT Mode: in this mode, the client sends the FTP server a PORT command followed by an IP address and a port number. On receiving this IP address and port number, the next time the server needs to transfer a file to the client, it will connect to this IP address and port number and transfer the file over that connection. The syntax the client sends will be something like "PORT 166,111,80,233,128,2".

From `http://cr.yp.to/ftp.html`:

> The PORT request passes the server a parameter in the form:
>
> h1,h2,h3,h4,p1,p2
>
> meaning that the client is listening for connections on TCP port p1*256+p2 at IP address h1.h2.h3.h4.
>
> The server normally accepts PORT with code 200. If the server was listening for a connection, it stops, and drops any connections already made. The server does not connect to the client's port immediately. After the client sends RETR and after the server sends its initial mark, the server attempts to connect. It rejects the RETR request with code 425 if the connection attempt fails; otherwise it proceeds normally."

PASV Mode: PASV is similar to PORT except that it is the server that specifies to the client the IP address and port number to connect to. When the client sends a PASV request to the server, the servers response will be something like "227 Entering Passive Mode (166,111,80,233,128,2)". The IP address you send to the client should be the IP address of the server. Choose a temporary random port number between 20000 and 65535 to send to the client. The server should then open a socket at that port number and listen on it.

From `http://cr.yp.to/ftp.html`:

> " RFC 959 failed to specify details of the response format. I recommend that servers use the format
>
> 227 =h1,h2,h3,h4,p1,p2
>
> where the server's IP address is h1.h2.h3.h4 and the TCP port number is p1*256+p2. The extra character before h1 is essential; otherwise old versions of Netscape will lose the first digit of h1.
>
> The server normally accepts PASV with code 227. Its response is a single line showing the IP address of the server and the TCP port number where the server is accepting connections.
>
> Normally the client will connect to this TCP port, from the same IP address that the client is using for the FTP connection, and then send a RETR request. However, the client may send some other requests first, such as REST. The server must continue to read and respond to requests while it accepts connections. Most operating systems handle this automatically.
>
> If the client sends another PASV request, the server normally accepts the new request with a new TCP port. It stops listening for connections on the old port, and drops any connections already made."

IP addresses and port numbers specified by PASV and PORT commands are one time use only. The client will need to make a PORT or PASV call before every file transfer. Otherwise, return an appropriate error

message.

From `http://cr.yp.to/ftp.html`:

> Some clients do not close the data connection until they receive the 226 response from the server. RFC 959 permits this behavior. (The intent, now obsolete, was for clients to retrieve multiple files through one data connection, with a self-delimiting encoding of each file. The server could use 226 to say that it was closing the connection, or 250 to say that it wasn't. The most obvious client implementation wouldn't close the connection until it received 226.) However, I recommend that clients close the data connection immediately after seeing the end of data. One server, wu-ftpd 2.6.0, waits until the client closes the connection before it sends its 226 response; this screws up file transfers to clients that do not close the data connection immediately. This also wastes a round-trip time for other clients. (As of 1999, various versions of wu-ftpd run about half of the Internet's FTP servers. Many servers made an emergency switch to version 2.6.0 in October 1999 when major security holes were discovered in previous versions.)
>
> In theory, the client can send RETR without a preceding PORT or PASV. The server is then supposed to connect to port 20 at the client's IP address. In practice, however, servers refuse to do this.

3.3 RETR AND STOR

FTP is used to transfer files. RETR is used by client to retrieve specific files from a server while STORE is used to store a specific file on a server. You will need to implement both. You can assume that all transfers will be done in binary mode. The syntax for RETR is "RETR <filename>".

From `http://cr.yp.to/ftp.html`:

> A RETR request asks the server to send the contents of a file over the data connection already established by the client. The RETR parameter is an encoded pathname of the file.
>
> Normally the server responds with a mark using code 150. It attempts to send the contents of the file over the data connection, and closes the data connection. Finally it
>
> - accepts the RETR request with code 226 if the entire file was successfully written to the server's TCP buffers;
> - rejects the RETR request with code 425 if no TCP connection was established;
> - rejects the RETR request with code 426 if the TCP connection was established but then broken by the client or by network failure; or
> - rejects the RETR request with code 451 or 551 if the server had trouble reading the file from disk.The server is obliged to close the data connection in each of these cases.
>
> The client is not expected to look for a response from the server until the client sees that the data connection is closed.
>
> The server may reject the RETR request without first responding with a mark. In this case the server does not touch the data connection. RFC 959 allows code 550 for file-does-not-exist, permission-denied, etc., and code 450 for out-of memory, disk-failure, etc.

STOR is similar.

When sending or receiving files in response to RETR and STOR commands from a client, you should ignore control commands from that particular client until the data connection is completed (if you want,

you can still handle ABOR / QUIT requests, but you must ignore everything else). However, new clients should still be able to connect to the server and existing clients should be able to start their own data transfers concurrently.

3.4 SYST, TYPE, QIUT, ABOR

You will need to handle the above 4 commands as follows

SYST: on receiving a SYST command, return the string "215 UNIX Type: L8" to the client.

TYPE: on receiving a "TYPE I" command, return "200 Type set to I." to the client. On receiving some other TYPE command, return an appropriate error code.

QUIT: On receiving a QUIT command, you will need to return the appropriate acknowledgement code to the client (example shown in section 4) and then log the client out. This will involve closing all the network connections involving that particular client. If you want, you can monitor some statistics like how many bytes a client transferred and report them back to the client before you close the connection (see example in section 4). But this is strictly optional.

ABOR: Abort is similar to QUIT. On receiving an ABOR command, process it as you would a QUIT command.

3.5 Request Strings/Marks

A mark is part of the reply string sent by the server to the client. It is sent in response to a request from a client.

Request format

A request is a string of bytes. It contains

1. a verb consisting of alphabetic ASCII characters;
2. optionally, a space followed by a parameter;
3. \r\n

For example, the following request (shown without the \r\n) contains verb RETR with parameter report.pdf:

RETR report.pdf

You can assume that the verb part of the request will be in capital letters (RETR, STOR etc.) but the optional parameter portion can be in any combination of upper and lower case characters (UNIX filenames are case sensitive so this matters for the filename parameter for RETR and STOR commands).

Response format

- The server's response consists of one or more lines. Each line is terminated by \r\n.
- The client can identify the last line of the response as follows: it begins with three ASCII digits and a space; previous lines do not. The three digits form a code. Codes between 100 and 199 indicate marks; codes between 200 and 399 indicate acceptance; codes between 400 and 599 indicate rejection.

For example, the following six lines have two responses:

150-This is the first line of a mark
123-This line does not end the mark; note the hyphen
150 This line ends the mark
226-This is the first line of the second response
226-This line does not end the response; note the leading space
226 This is the last line of the response, using code 226

Servers are required to follow several more rules. Each line of the response is required to contain \r immediately before the terminating \n. If the answer has more than one line, its first line is required to begin the same way as the last line, but with a hyphen in place of the space.

RFC 959 prohibited all codes other than 110, 120, 125, 150, 200, 202, 211, 212, 213, 214, 215, 220, 221, 225, 226, 227, 230, 250, 257, 331, 332, 350, 421, 425, 426, 450, 451, 452, 500, 501, 502, 503, 504, 530, 532, 550, 551, 552, and 553. (Typically the second digit is 0 for a syntax error, 1 for a human-oriented help message, 2 for a hello/goodbye message, 3 for an accounting message, or 5 for a filesystem-related message.) However, clients cannot take this list seriously; the IETF adds new codes at its whim. Hence, many clients avoid looking past the first digit of the code, either 1, 2, 3, 4, or 5. The other two digits, and all other portions of the response, are primarily for human consumption. (Exceptions: Greetings, responses with code 227, and responses with code 257 have a special format.)

Servers must not send marks except where they are explicitly allowed. Many clients cannot handle unusual marks. Typical requests do not permit any marks.

Some clients are unable to handle responses longer than one line to various requests, even though RFC 959 permits multiple-line responses under most circumstances. The client we are using for testing supports multi line responses in some cases (see example in Section 4). Hence, use one-line responses whenever in doubt.

The server can reject any request with code

- 421 if the server is about to close the connection;
- 500, 501, 502, or 504 for unacceptable syntax; or
- 530 if permission is denied.

Typically 500 means that the request violated some internal parsing rule in the server, 501 means that the server does not like the format of the parameter, 502 means that the server recognized the verb but does not support it, and 504 means that the server supports the verb but does not support the parameter.

RFC 959 states that codes 400 through 499 are for temporary errors, and codes 500 through 599 are for permanent errors.

3.6 MULTI-CLIENT SUPPORT

An FTP server that accepts only one connection at a time is probably impractical and definitely not very useful. As such, your server should also be written to accept multiple connections (usually from multiple hosts). It should be able to simultaneously listen for incoming connections as well as keep reading from whatever connections are already open.

Unfortunately, many of the calls you will use (eg. accept(), recv()) are blocking calls, that is, they go to sleep and stall program execution until some data arrives. So if your server is blocking on an accept() call, for instance, it cannot receive() data at the same time. This could lead to starvation of certain clients, i.e. they never get served. One solution to this problem would be to use a call such as fcntl(sockfd, F_SETFL, O_NONBLOCK)to set the socket to a non-blocking one, and then poll for information. Generally speaking, putting your program on busy-wait loop looking for data on a socket consumes enormous amounts

of CPU time and is a bad idea.

select(), on the other hand, gives you the power to monitor several sockets at the same time, blocking until there is data to be dealt with. It will tell you which ones are ready for reading, writing, and which have raised exceptions (if you really want to know).

Your server must deal with multiple connections. A single threaded server is entirely adequate if you choose to use the select() call to implement multiple connections on your server. Alternatively, it is perfectly fine if you prefer to use multiple threads (using pthreads) or multiple processes (using fork) to implement your own connection management.

4 EXAMPLE

Following is a log of an actual session from a real FTP client to a real FTP server. This session incorporates all the commands that you will need to implement and shows typical response strings from the server. Please review it to see how an FTP server is supposed to respond to a client.

1. Client connects to the server (ftp.ssast.org)
2. Server responds with an initial message
   "220 ftp.ssast.org FTP server ready."
3. Client then attempts to log in by sending
   "USER anonymous"
4. Server parses the argument, determines it is open and requests the client for the password with the following response
   "331 Guest login ok, send your complete e-mail address as password."
5. Client responds with the email address as the password
   "PASS dangfan@163.com"
6. Server determines that the username and password are acceptable. It logs the client in and displays the welcome message. Notice that only the last line of the welcome message contains a valid mark as explained in section 3.5

   230-
   230-Welcome to
   230- School of Software\r\n
   230- FTP Archives at ftp.ssast.org\r\n
   230-\r\n
   230-This site is provided as a public service by School of\r\n
   230-Software. Use in violation of any applicable laws is strictly\r\n
   230-prohibited. We make no guarantees, explicit or implicit, about the\r\n
   230-contents of this site. Use at your own risk.\r\n
   230-\r\n
   230 Guest login ok, access restrictions apply.\r\n

7. Client tries to determine the servers operating system and type settings by sending a SYST command
   "SYST"
8. Server responds with its SYST settings
   "215 UNIX Type: L8"
9. Client decides to set the TYPE to binary (type I)
   "TYPE I"
10. Server responds that the operation was successful
    "200 Type set to I."

11. Client sends the PORT command to the server
    "PORT 166,111,80,233,128,79"
12. Server responds that it acknowledges the PORT command
    "200 PORT command successful."
13. Client attempts to retrieve a file (robots.txt)
    "RETR robots.txt"
14. Server opens a binary connection to the IP address and port number specified by the earlier PORT command
    "50 Opening BINARY mode data connection for robots.txt (26 bytes)."
15. Server tells the client that the transfer is complete
    "226 Transfer complete."
16. Client decides to tell the server to use PASV mode instead
    "PASV"
17. Server responds with the IP address and port number for the client to connect to
    "227 Entering Passive Mode (166,111,80,233,102,109)"
18. Client retrieves the same file again (robots.txt) "RETR robots.txt"
19. Server accepts a connection from the client to the IP address and port number specified by the PASV command. It sends the file over in binary mode. Note that the messages that the server returns are identical in both PORT and PASV mode.
    "150 Opening BINARY mode data connection for robots.txt (26 bytes)."
20. Server tells the client that the transfer is complete
    "226 Transfer complete."
21. Client decides to logout
    "QUIT"
22. Server logs the client out and displays some statistics about the ftp connection

    221-You have transferred 52 bytes in 2 files.
    221-Total traffic for this session was 1975 bytes in 2 transfers.
    221-Thank you for using the FTP service on ftp.ssast.org.\r\n
    221 Goodbye.\r\n

5 Helpful Hints

- We provide several test cases in the issued zip file, including 28 credits for server test and 22 credits client test. The remaining test cases will be not be made public.
- You do not need to worry about file permissions. When executing RETR and STOR commands, if you get errors when opening the necessary file descriptors, just return an appropriate error code to the client (this assumes of course that your code for opening the file descriptors is correct.)
- You can assume that the FTP server is running on a **Linux** server, so your client must to be able to parse at least linux-formatted directories.
- Low port numbers are reserved for system services and such. As far as possible, try to use port numbers between 20000 and 65535.
- Make sure you always prepend the root directory specified with the –root option (defaults to /tmp if not specified) to file requests from clients.
- For security reasons, you should not accept RETR or STOR requests having the pattern "../" as part of their parameter. It is in your own interest to implement this simple access control policy, since you do not want random people to be able to read or overwrite personal files in your home directory.
- Files that you might need to #include are
  - <sys/types.h>
  - <netinet/in.h>

  - <arpa/inet.h>
  - <netdb.h>

  Other useful files to include are

  - <fcntl.h> (if you need to set any of the sockets to be non-blocking)
  - <sys/select.h> (if you are using select)
  - <pthread.h> (if using pthreads)
  - <stdio.h> (for printf etc)
  - <errno.h> (for using the errno feature of c)
  - <strings.h> (for all the string functions)

- In case you have difficulties, you can contact the TAs by email: ycdfwzy@outlook.com, zgxw18@outlook.com, xushen20@mails.tsinghua.edu.cn or visit the TAs at 11-211, East Main Building.