Basic FTP Client and Server

Build a network client and server that implements the "basic ftp" protocol as documented in this assignment.

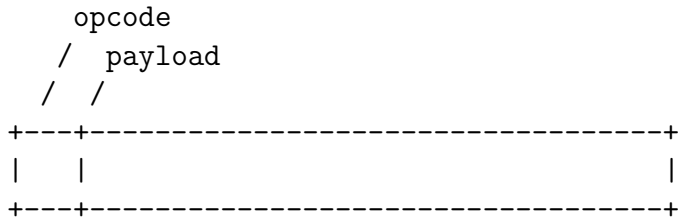
Protocol

The basic ftp protocol has 5 message types as follows:

Type	Description
GET	request to download a file from the server
PUT	request to upload a file to the server
DAT	a data packet
RDY	server is ready to process the client's request
ERR	something is wrong with the client's request

The type (a.k.a. opcode) in the message is **ALWAYS** upper case.

The packet format for each message type is as follows:



- Opcode, 3 bytes, ASCII, **NOT** null terminated
- Payload, 512 bytes, ASCII, Null terminated when **NOT** a data packet

The opcode is not null terminated. If the payload is a file name or error message, it is null terminated. Otherwise the payload is **not** null terminated (i.e. a DAT packet).

The end of a file transfer is signalled by DAT packet with less than 512 bytes of payload. All data packets are type DAT and must have exactly 512 bytes of payload except for the final packet. The final DAT packet must have zero to 511 bytes of payload. Note that this means if the file size being transferred is a multiple of 512, the server must send a final DAT packet with a zero length payload to signal the end of the transfer.

Notice that the entire packet is ASCII (plain text, not binary). This is conventional with most Internet protocols.

You must create a separate *C* header and implementation for the "basic ftp" protocol. Both your server and client must use (include) this protocol. Your protocol design must implement at least the following features:

Required operation	Description
create a new packet	creates a new message packet from parameters
extract the message type	returns the message type (opcode) from a packet as a code or enumerated type
extract the message payload	extracts the payload from a packet and copies it into a buffer

Client

The client requires one command line argument. The argument to the client is the hostname of the basic ftp server. If this argument is not provided the client should display an "usage" message and terminate.

Once the client is running, it becomes interactive (like a shell). The user will type a command and the client will perform the requested operation. There are four possible commands (operations): GET, PUT, BYE, and HELP. Any combination of case must work for the commands.

The GET and PUT commands require a file name. The file name is case sensitive (i.e. the case must match). The GET command will cause the client to download the named file from the server. The PUT command will cause the client to upload the named file to the server.

If the client is unable to complete a operation, then it will send an ERR message to the server and display an error to the user (for example "error: unable to open foo.txt"). The client does not terminate when it sends or receives an ERR message, it will wait for another command from the user.

The HELP command will display a brief description of the client commands.

```
printf("Basic-ftp commands:\n"
       "\t GET - download the named file from the basic-ftp server\n"
       "\t PUT - upload the named file to the basic-ftp server\n"
       "\t BYE - quit\n"
      );
```

The BYE command will cause the client to clean up and terminate. The client may transfer several files in a single session (multiple PUT and/or GET commands) before disconnecting with a BYE command.

The client always starts the protocol and may make multiple requests per session. Figure 1 shows a client session with a GET command and a data transfer, followed by the client sending a BYE command.

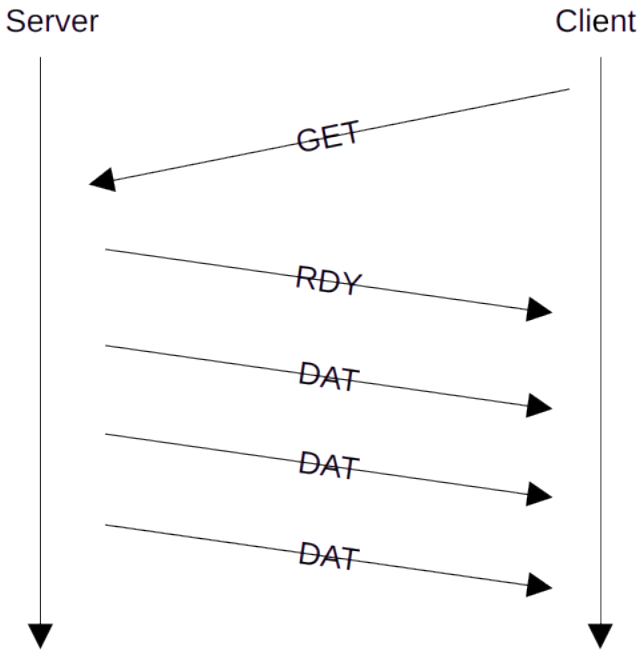


Figure 1: Example basic ftp protocol message exchange

Server

The server has no command line input and no console output. The server should continue waiting for new clients after each client session is completed. The server should only terminate when a SIGTERM is received. The server must terminate immediately when it receives SIGTERM. The server must use signal(3) and select(2).

The server will respond with RDY to acknowledge a PUT or GET if it is ready to process the request (no errors). The server will send an ERR message if it is unable to fulfill the PUT or GET request. The ERR packet should include an informative error string (null terminated) as the payload.

- File not found
- File access denied
- Unable to write file

★ Your server and client **MUST** be located in different folders when testing.
Run the client from a different folder when testing to avoid file contention. In other words, you cannot download a file from a folder into the same folder. Also, you cannot upload a file to a folder from the same folder.

Put the client and server in separate folders for testing.

Your programs must be developed in *C* (-std=c17) using the BSD socket API on the CS server. The programs must compile and run on the CS server with cc (clang).

★ Note: To avoid conflicting with your classmate's programs when testing on the CS server, use the last three digits of your student ID L# plus 11000 for your port number.

Due dates:

Friday 19th before 11:59 p.m.
Completed server commit, tag it "server"

Friday 26th before 11:59 p.m.
Completed client commit, tag it "client"

Use an annotated tag *after* your commit to tag the commit.

How to submit:

Create an empty git(1) repository in a folder named "program-4". Commit and tag your files as described above in "Due dates". You may, and should, commit more frequently than the required commit dates. You should build your server and client programs in stages (skeleton, startup, etc.) and commit the working version of each stage as you complete it.

Program evaluation:

Your program must conform to the **UNA C Code Style**. The "UNA C Code Style" is linked on the class web page. Code will be evaluated for correctness, efficiency, and style.

Helpful resources:

```
Command line arguments: https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/
--
man pages - err(3), printf(3), scanf(3)
            fopen(3), fwrite(3), fread(3), fclose(3)
            memset(3), memcpy(3), strlcpy(3)
            style(9), nc(1)
```

Additional resources:

```
https://beej.us/guide/bgnet/
https://csrc.nist.gov/Projects/ssdf
https://sourceware.org/gdb/current/onlinedocs/gdb.html
```