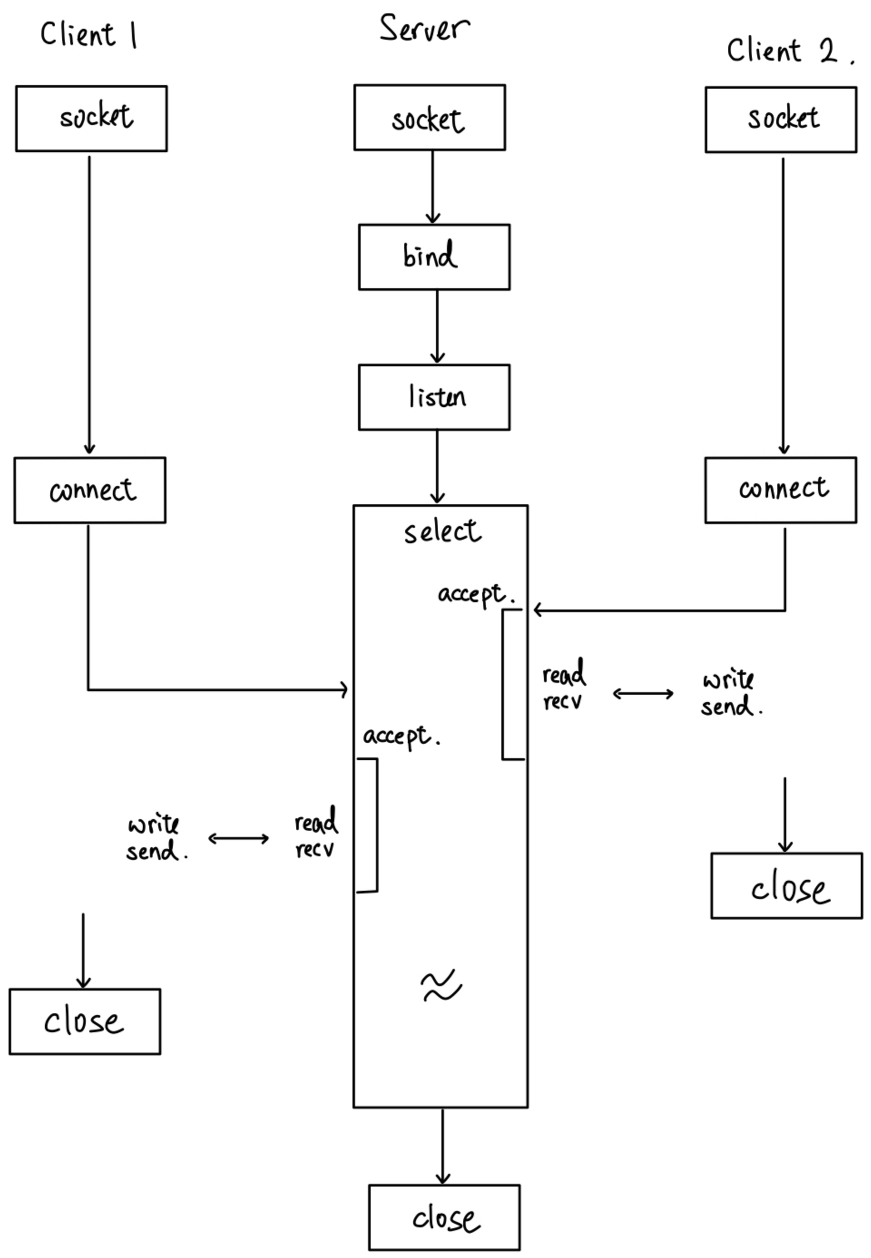
Flowchart of the connection:



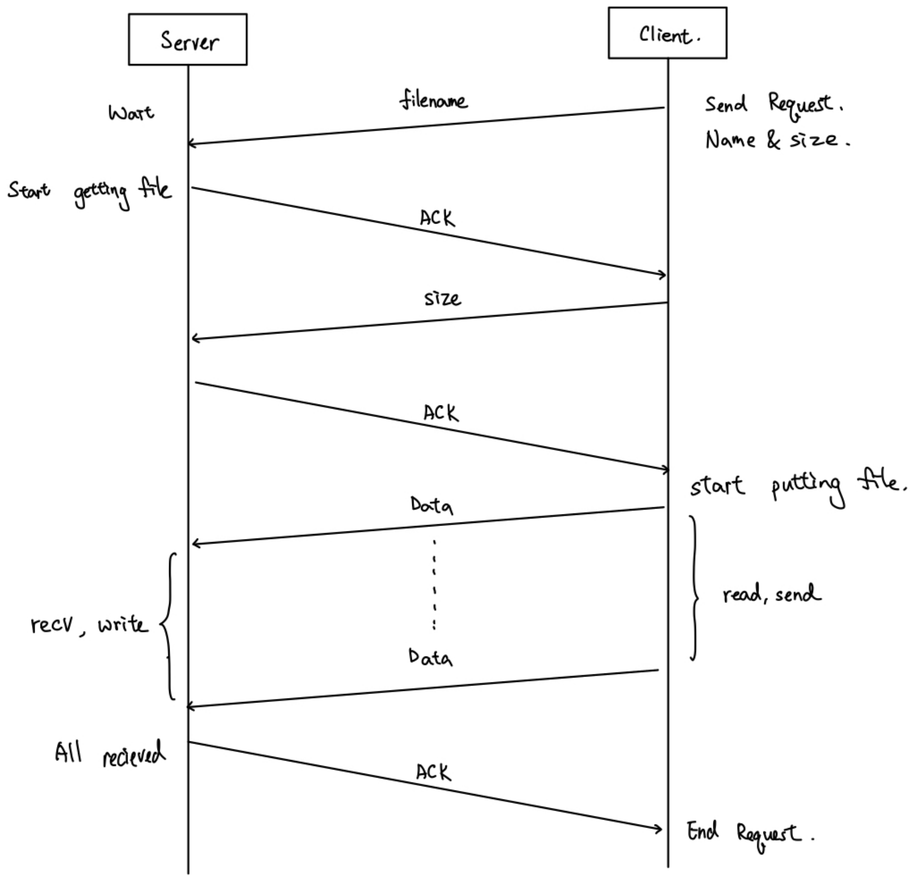
First, server will set up the socket with 3 functions call including *socket, bind, listen.* Client will also set up with a single call *socket*. Then, socket will call *select* to wait for the clients to send request to socket. Meanwhile, client will call *connect* to connect to server's listening socket.

Whenever there a client called *connect* to connect to the server, server will notice by *select*. The server's socket listening file descriptor will be set by *select* and then server will then call *accept* to assign a new socket to the connected client.

Then later on, socket will constantly call *select* to look for new connection and connected server to write data. The situation when new connection occur is stated above. However, when some client (previous assigned socket) is ready to write command to the server, its read file descriptors set will be set by *select*.

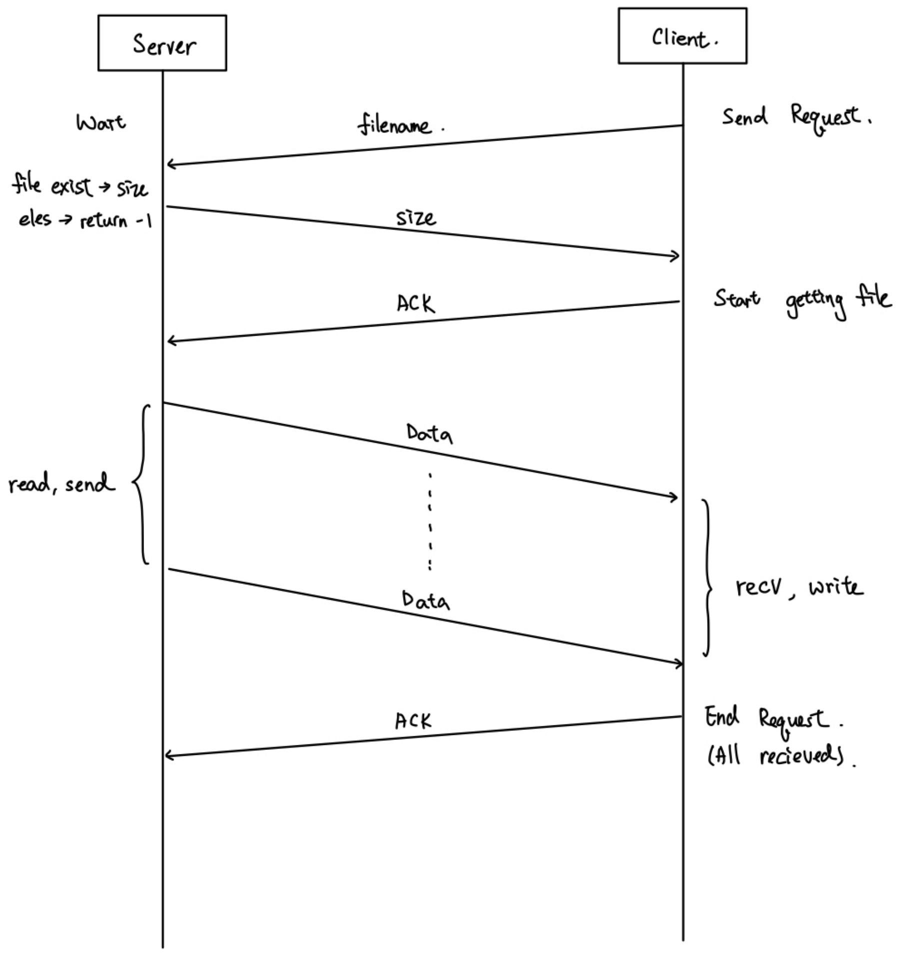
Then, the server will linearly go through ever assigned socket to check which socket is ready to perform task. Thus, by doing so, only needed socket will be handled by the server and thus we can handle multiple user at the same time. We can perform I/O multiplexing and *select* will tell which client is ready to be read, then further file transmitted or list can be performed.

Flowchart of client's put file:



Client will send filename to the server and server will send back ACK after receive filename. Then, the client will find the size of the file and the rest is history. Then, the data will be transmitted to server through read and send from client and recv and write to server. Since the recv and send might not be 1 to 1 (etc. multiple send might be read by 1 send), we have to count the received file size to know whether the file is finished being transmitted. At last, server will send an ACK to tell client that the file is transmitted successfully. Thus, the whole put file process is done and server can handle the next connection.

Flowchart of client's get file:



Client will send filename to the server and server will send back the size of the. Then, the data will be transmitted to server through read and send from server and recv and write to client. Since the recv and send might not be 1 to 1 (etc. multiple send might be read by 1 send), we have to count the received file size to know whether the file is finished being transmitted. At last, client will send an ACK to tell server that the file is transmitted successfully. Thus, the whole put file process is done and client can handle the next connection.

What is SIGPIPE? It is possible to happen to your code? If so, how do you handle it?

SIGPIPE is a broken pipe signal and happened when someone write or read to broken or ended pipe (or socket).

When the client ends the process (etc. ^c), the server will receive a SIGPIPE signal and terminate; since the signal routine for SIGPIPE will kill the process who received it.

So, in order to avoid server being closed by client, I register the signal handler of SIGPIPE to prevent it from terminating the process with *signal*. Thus, when the server receives the SIGPIPE signal, it will clear the file descriptor and reinitialize the user state.