File Server Implementation

Summary: In this homework, you will be implementing a pair of programs that simulate Server-Client interaction on a single computer using Socket programming.

1 Background

In this assignment you will write a program that simulates client-server interaction using Socket Programming. Since, there is not enough description about these ideas in the textbook, plan to devote some time to understanding it. The textbook's coverage is contained in Section 3.6 (Communication in Client-Server Systems). When we talk about client-server systems, we are talking about networking, i.e. communication between the devices connected over a network. One way to understand networking is through the diagram below:

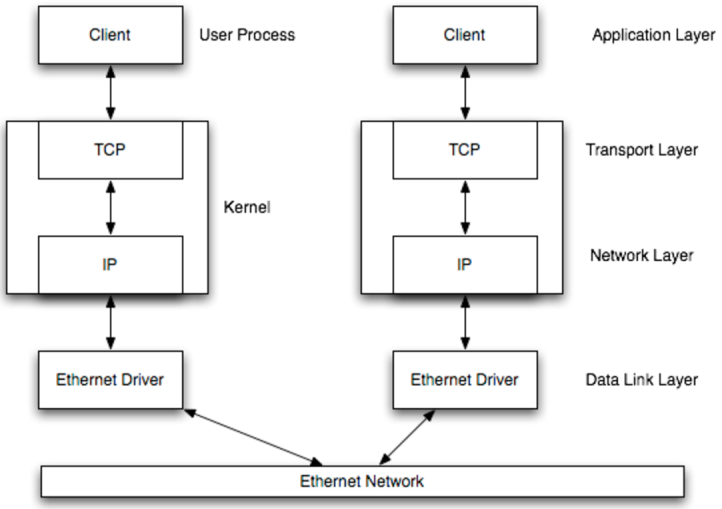


Figure 1: Image from CS60 (A. T. Campbell).

Data is passed starting from the application layer of the sending process, where at each layer data is encapsulated within the header provided by the layer and forwarded to the lower layers which transmit data to the recipient through physical links or wirelessly. Here, the application refers to the processes which initiate the transmission of data over the network, for example: mail server, FTP server, HTTP server etc. Although each network layer has their own significance, the two most important layers from the viewpoint of operating system are the Transport Layer and Network Layer collectively known as TCP-IP layer based on the protocols implemented in them respectively.

Talking about what these layers do, it becomes obvious to wonder how data is able to reach its destination! Just like there is an address associated with the postal mail, there is one associated with the data transmitted over a network. This information is known as an *IP address*, and associated with Network layer, is the primary means which enables data to figure out its destination in the network as every device connected to a network has its own IP address. Now, once the data reaches the target machine or address over a network, it is necessary to somehow determine the exact process or the application to which that data is to be delivered because at a high-level this whole process is about the interaction between two processes like that between

an email client on one machine and email server on another machine. So, if IP address is the address of a building, then the *PORT number* (associated with Transport layer) is the room number in that particular building.

This is where **Sockets** come into picture. Sockets are defined as endpoints for communication. A pair of processes communicating over a network employ a pair of sockets - one for each process. A socket is identified by an IP address concatenated with port number. Implementing a client-server architecture involves creating a socket, listening at a particular port, acting as a server and waiting for incoming client requests. Once a request is received, the server accepts a connection from the client socket to complete the connection. The diagram below specifies the sequence of methods to be implemented to establish a connection between two processes also known as client-server.

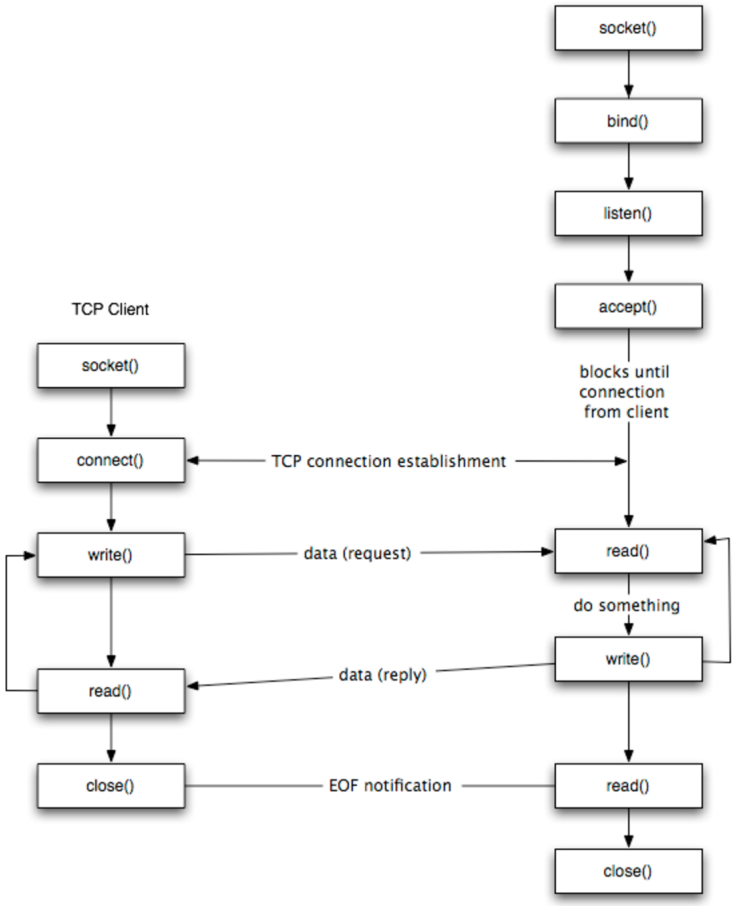


Figure 2: Image from CS60 (A. T. Campbell).

This document is separated into four sections: Background, Requirements, Header Files, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Header Files, we discuss several header files that include functionality related to file and directory manipulation. Lastly, Submission discusses how your source code should be submitted on BlackBoard.

2 Requirements [35 points]

In this assignment you are required to create a file server (20 points) and client (15 points), on the port number specified as an argument. As a base requirement, your program must compile and run under GCC

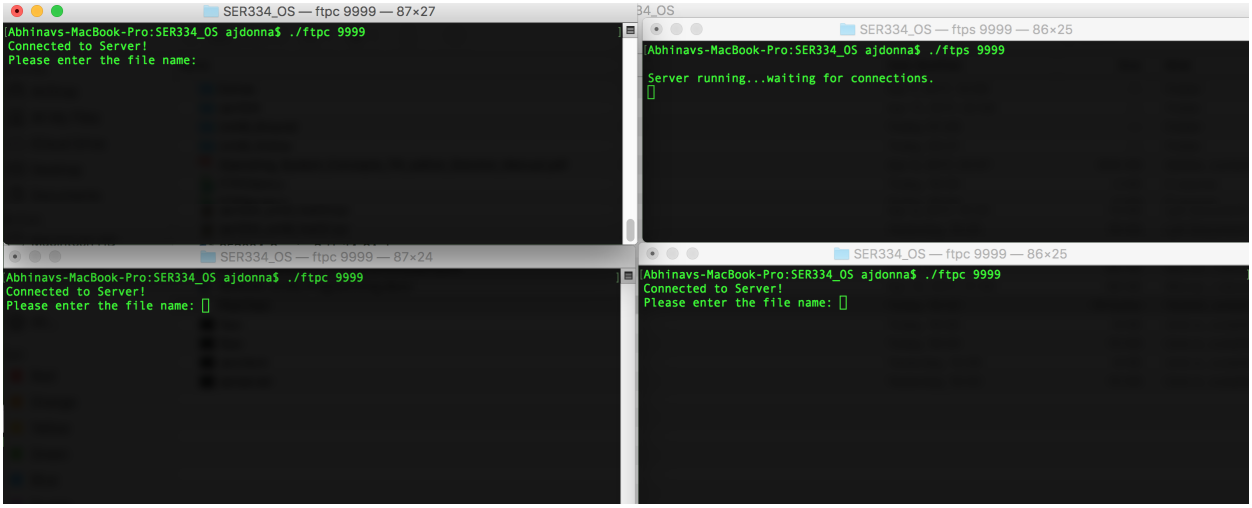
on Xubuntu (or another variant of Ubuntu) 16.04. The file server should:

- Listen to the incoming connections on a port number specified as a command line argument.
- Be capable of handling incoming requests in parallel (at most 5) with the concept of pthreads.

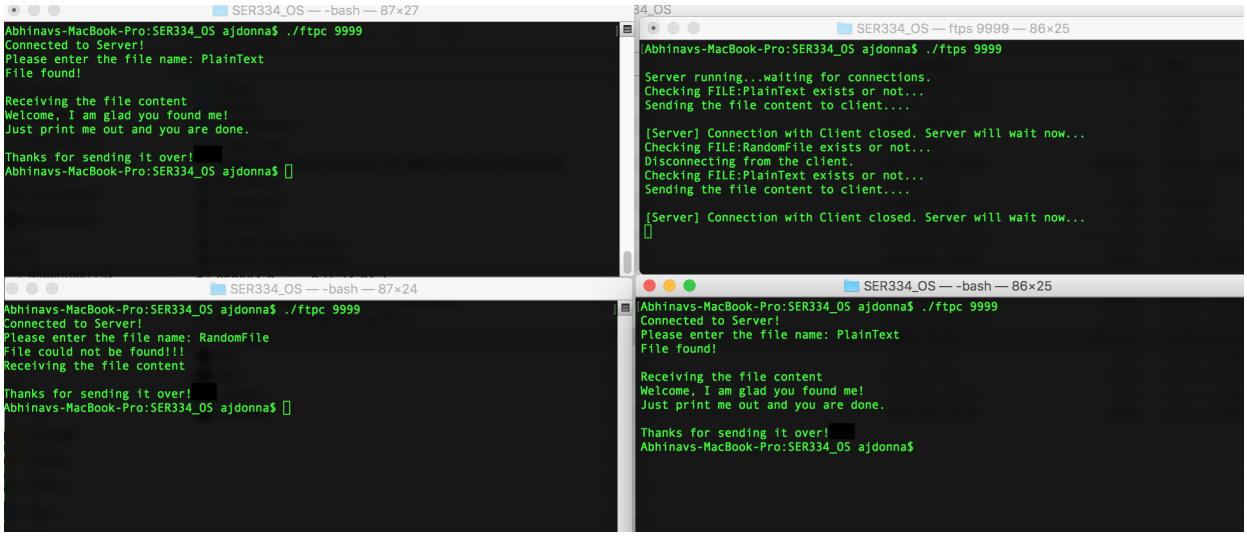
The client should:

- Connect to localhost on a port number specified as a command line argument.
- Offer to transfer exactly one file.

The overall process will be that the client will provide a filename to the server and the server will then open that file, read the contents of the file, and send the data back to the client. Assume that the file being transferred will be a text file with at most 512 characters on each line. If the file doesn't exist, the server will send back a message saying "File Not Found". If the file exists, then the client will create a local copy of that file in the directory the program is started. Please see the attached snapshots for details:



As visible above, the server **ftps** running on port **9999** is waiting for the incoming connections and three clients are connected to server simultaneously, being prompted to enter the file name to look up by the server. On entering the file name, the response should be:



Now, as visible in the top left portion the server sends back the client content of the requested file, whereas in the bottom left it is visible that server sends the message **"File could not be found"!** The rest of the content appearing is just for debugging purposes. **Imp:** The input should be provided as specified above i.e. **PORT Number** being specified at the command line and the **IP address** by default is localhost i.e. **127.0.0.1.** You are not required to exactly mimic the messages being shown above but there should be proper messages for Client Connected/Not Connected, File Found/Not Found, File Content and Connection getting closed. Moreover, server should keep running until it is intentionally closed. By that it means that after serving 5 clients or any other existing condition, server should not close automatically!

Implementing both the client and server should not be too daunting because the structure of both would be based on the methods (bind(), listen(), accept(), send(), receive()) outlined above. It's just that the server would have to perform additional processing of incoming connections and file handling.

3 Header Files

The below functions and header files are the primary resources that would help us in successful implementation of the program.

3.1 Functions

- **socket() - int socket(int domain, int type, int protocol).** The socket() function creates an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets. (By default we use SOCK_STREAM with TCP connections). Upon successful completion, socket() returns a non-negative integer, the socket file descriptor. Otherwise, a value of -1 will be returned.
  - **domain** argument specifies the address family used in the communications domain. The address families supported by the system are implementation-defined.
  - **type** argument specifies the socket type, which determines the semantics of communication over the socket. The following socket types are defined:
    - **SOCK_STREAM** Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.
    - **SOCK_DGRAM** Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.
    - **SOCK_SEQPACKET** Provides sequenced, reliable, bidirectional, connection-mode transmission paths for records.
- **bind() - int bind(int socket, const struct sockaddr *address, socklen_t address_len)**. The bind() function assigns a local socket address address to a socket identified by descriptor socket that has no local socket address assigned. Sockets created with the socket() function are initially unnamed; they are identified only by their address family. Upon successful completion, bind() returns 0; otherwise, -1 wiill be returned and errno set to indicate the error. The bind() function takes the following arguments:
  - **socket** Specifies the file descriptor of the socket to be bound.
  - **address** Points to a sockaddr structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
  - **address_len** Specifies the length of the sockaddr structure pointed to by the address argument.
- **listen() - int listen(int sockfd, int backlog).** listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests.
  - **sockfd** File descriptor that refers to a socket of type SOCK_STREAM or SOCK_SEQPACKET.
  - **backlog** Maximum length to which the queue of pending connections for sockfd may grow.

- **accept() - int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen).** The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.
  - **sockfd** The argument sockfd is a socket that has been created with socket(), bound to a local address with bind(), and is listening for connections after a listen().
  - **addr** The argument addr is a pointer to a sockaddr structure.
  - **addrlen** The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr;
- **connect() - int connect(int socket, const struct sockaddr *address, socklen_t address_len).** The connect() function will attempt to make a connection on a socket:
  - **socket** Specifies the file descriptor associated with the socket.
  - **address** Points to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket.
  - **address_len** Specifies the length of the sockaddr structure pointed to by the address argument.

3.2 Header Files

- **<sys/socket.h>:** Includes a number of definitions of structures needed for sockets such as
  - **socklen_t**, which is an unsigned opaque integral type of length of at least 32 bits
  - **sockaddr** structure, which is defined as below:
    - struct sockaddr_in {
      - short sin_family; /*Set to AF_INET*/
      - u_short sin_port; /*Set to the server's port number*/
      - struct in_addr sin_addr; /*Set to the server's address*/
      - char sin_zero[8]; /*unused*/ };
- **<sys/types.h>:**This header file contains definitions of a number of data types used in system calls used by sockets.
- **<netinet/in.h> and <arpa/inet.h>:** This header file contains constants and structures needed for internet domain addresses. The most important being Host Byte Order to Network Byte Order Conversion. There are two ways to store two bytes in memory: with the lower-order byte at the starting address (little-endian byte order) or with the high-order byte at the starting address (big-endian byte order). We call them collectively host byte order. For example, an Intel processor stores the 32-bit integer as four consecutive bytes in memory in the order 1-2-3-4, where 1 is the most significant byte. IBM PowerPC processors would store the integer in the byte order 4-3-2-1. The header file provides below main functions:
  - uint16_t htons(uint16_t host16bitvalue);
  - uint32_t htonl(uint32_t host32bitvalue);
  - uint16_t ntohs(uint16_t net16bitvalue);
  - uint32_t ntohl(uint32_t net32bitvalue);
- **<pthread.h>:** The header file dealing with the methods associated with pthreads in C.

4 Submission

The submission for this assignment has one part: a source code submission. The file should be attached to the homework submission link on BlackBoard.

Writeup: For this assignment, no write up is required.

Source Code: Please name your in two files as "LastNameServer.c" and LastNameClient.c" (e.g. "JainServer.c" and "JainClient.c").