SIMPLE MAIL TRANSFER PROTOCOL

ASSIGNMENT 1

TEENA K
CS15B036

CONTENTS

TASK: To implement a simple email client, and a mail server using the Socket Interface.

CLIENT

A client is a process that makes a service or information available by requesting a server.

SERVER

Server is a process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client.

SOCKET PROGRAMMING

Socket programming involves connecting a server and its clients for information sharing using Sockets. A socket is an junction of two-way communication link between two programs running on the network. A socket is bound to a port number.

In socket programming, a server runs on a machine and has a socket bound to a specific port. The server has to listen to the socket, in order to connect to a client making a request connection with this server. After the server accepting the connection, A new socket bound to a new port is given to the server, so that it can continue to listen to the orginal socket for connection requests while serving the connected client.

TCP :

Transmission Control Protocol TCP provides transferring of data between applications running on hosts communicating by an IP network **.** A connection between two computers uses a socket.

FUNCTIONS:

The header "sys/socket.h" is included in the files, to avail all the functions related to sockets.

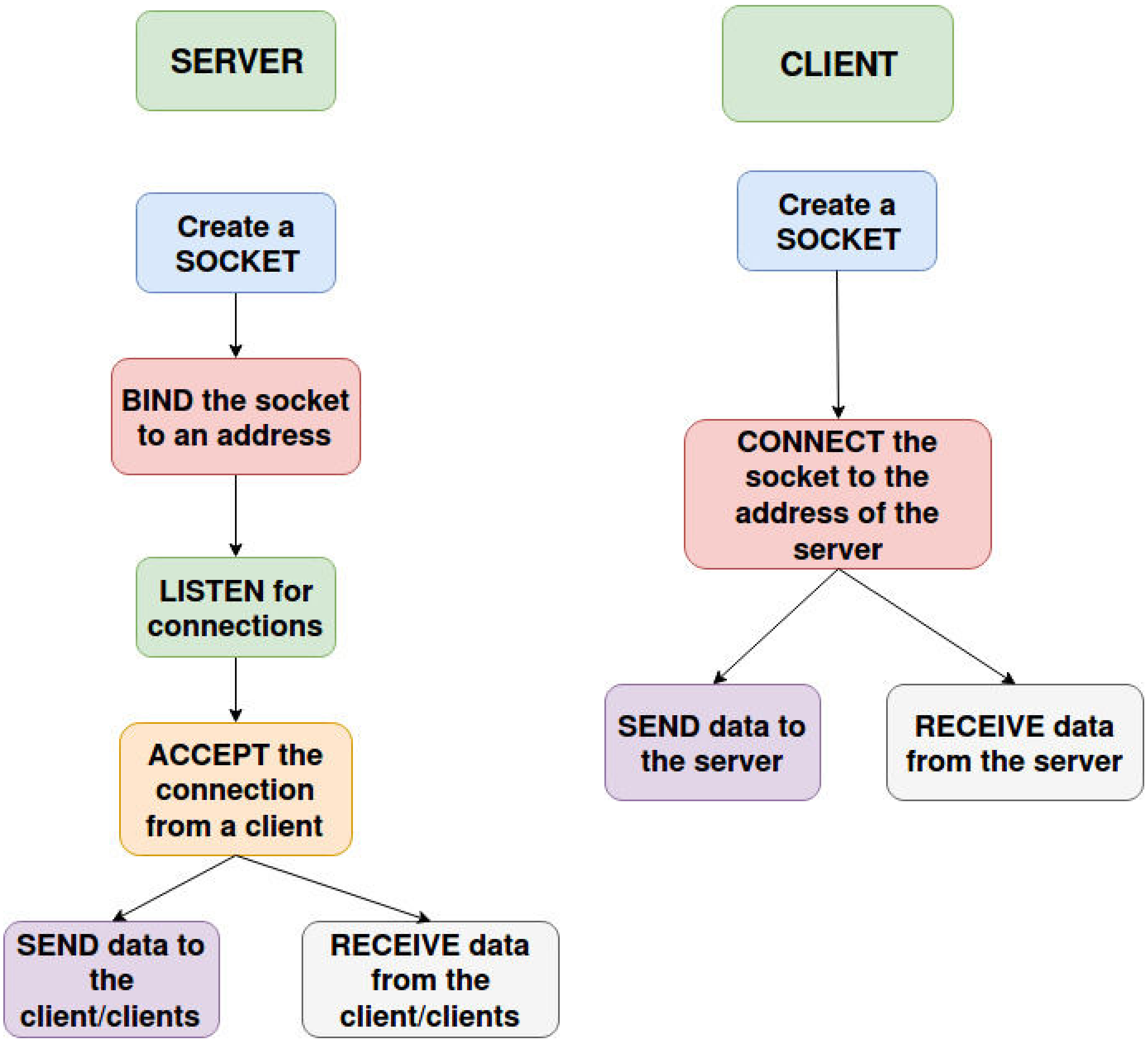
- ***Creating a Socket:***

int socket(int *domain*, int *type*, int *protocol*);

The *socket*() function creates an unbound socket, and returns a file descriptor that can be used in later function calls that operate on sockets. The *socket()* function takes the following arguments:

- *domain*: The *domain* argument specifies the address family used in the communications domain
- *type:* Specifies the type of socket to be created.
- *protocol:* Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket*() to use an unspecified default protocol appropriate for the requested socket type.

Return Value: Upon successful completion, *socket*() returns a non-negative integer, the socket file descriptor. Otherwise, a value of -1 is returned.


SERVER
Create a SOCKET
BIND the socket to an address
LISTEN for connections
ACCEPT the connection from a client
SEND data to the client/clients
RECEIVE data from the client/clients
CLIENT
Create a SOCKET
CONNECT the socket to the address of the server
SEND data to the server
RECEIVE data from the server

- ***Binding the Socket:***

  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.
  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.

  Return Value: Upon successful completion, *bind*() returns 0; otherwise, -1 is returned

- ***Listening for connection:***

  int listen(int *socket*, int *backlog*);

  The *listen*() function marks a connection-mode socket, specified by the *socket* argument, as accepting connections.
  The *backlog* argument is used to limit the number of outstanding connections in the socket's listen queue.

  Return Value: Upon successful completions, *listen*() returns 0; otherwise, -1 is returned.

- ***Accepting client connection request:***

  int accept(int *socket*, struct sockaddr *restrict *address*, socklen_t *restrict *address_len*);

  The *accept*() function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.

  The *accept*() function takes the following arguments:

  - *socket:* Specifies a socket that was created with *socket*(), has been bound to an address with *bind*(), and has issued a successful call to *listen*().
  - *address:* Either a null pointer, or a pointer to a **sockaddr** structure where the address of the connecting socket shall be returned.
  - *address_len:* Points to a **socklen_t** structure which on input specifies the length of the supplied **sockaddr** structure, and on output specifies the length of the stored address.

  Return Value: Upon successful completion, *accept*() returns the non-negative file descriptor of the accepted socket. Otherwise, -1 is returned.

- ***Connecting to server:***

  int connect(int *socket*, const struct sockaddr **address*, socklen_t *address_len*);

  The *connect*() function attempts to make a connection on a socket. The function takes the following arguments:

- *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.

  Return Value: Upon successful completion, *connect*() returns 0; otherwise, -1 is returned.

- ***Sending data:***

  ssize_t send(int *socket*, const void **buffer*, size_t *length*, int *flags*);

  The *send*() function shall send a message only when the socket is connected. The *send*() function takes the following arguments:

- *socket:* the socket file descriptor.
- *buffer:* to the buffer containing the message to send.
- *length:* Specifies the length of the message in bytes.
- *flags:* Specifies the type of message transmission.

  Return Value: Upon successful completion, *send*() shall return the number of bytes sent. Otherwise, -1 shall be returned.

- ***Receiving data:***

  ssize_t recv(int *socket*, void **buffer*, size_t *length*, int *flags*);

  The *recv*() function receives a message from a connection-mode or connectionless-mode socket. The *recv*() function takes the following arguments:

- *socket:* Specifies the socket file descriptor.
- *buffer*: Points to a buffer where the message should be stored.
- *length:* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.
- *flags:* Specifies the type of message reception.

  Return Value: Upon successful completion, *recv*() returns the length of the message in bytes. Otherwise, -1 shall be returned.

PROCEDURE:

The files server.c and client.c use the functions *socket(), bind(), listen(), connect(), accept()* to establish a connection between each other. After the connection is established, a series of *recv()* and *send()* functions are used to send and receive commands in orded to implement SMTP. The following commands are supported:

- ***Listusers***

  The client will send the message LSTU userid to the server, and print the server's response.

- ***Adduser <userid>***
  The client will send the message ADDU userid to the server, and print the server's response.
- ***SetUser <userid>***
  The client sends the message USER userid to the server, and prints the server's response.
- ***Read***: The client will send the message READM to the server, and print the server's response.
- ***Delete***: The client will send the message DELM to the server, and print the server's response.
- ***Send <receiverid>***: The client prints a prompt, "Type Message:". The user will type a mail message terminated by ###. The client then sends the message SEND receiverid <message> to the server, and prints the server's response.
- ***Done***: The client will send the message DONEU to the server, and print the server's response.
- ***Quit***: The client will send the message QUIT to the server, print the server's response, and close the TCP session.

The files are run accordingly:

- In terminal 1- *$gcc server.c -o server*
- In terminal 1- *$./server*
- In terminal 2 - *$gcc client.c -o clilent*
- In terminal 2 - *$./client*

RESULTS: As the files are run, and users are added, the server creates spool files, for each new user in the root directory – MAILSERVER. It is from these spool files of each user, that the server performs the actionss – read, send, delete as per the wishes of the client. The changes are accordingly updated in the spool files of the corresponding users.