HTTP Server and File Upload using Socket Programming in C

Assignment Report

CS703 : Network Engineering

Author
Sourabh Jain
16CS21F

Department of Computer Science and Engineering
National Institute of Technology, Karnataka

Contents

List of Figures

1 Socket Programming

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

The list of steps should be done at server and client side to initiate a connection. The flow chart below depicts the steps.

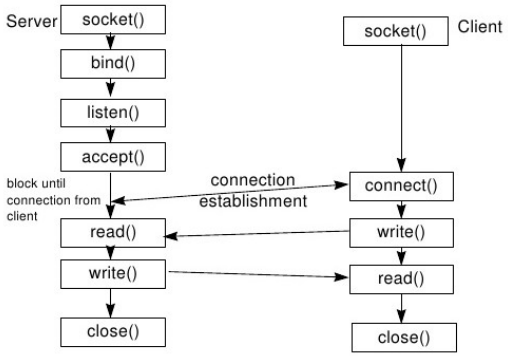

Figure 1: Client-Server Flow Chart

Server creates a socket(end point of communication) by using socket fuction and binds it with a port number. Once binding is done it puts that socket on listening mode. Now server is in waiting state, waits for a client to connect. Client also create a socket on their machine and by using connect function it begins communication with server. Once client request reached server machine it accept the connection and allocate the the resources for this connection.

2 Application 1 : Echo Server

2.1 Objective

To create a server that reposed to client with same message that sends by a client.

2.2 Server Implementation

To implement a Echo Server few basic libraries are required that provides function to create socket, send message, and get the message fom file descriptor.

Basic input output library

```
#include <stdio.h>
#include <stdlib.h>
```

To use basic networking functions like:- socket, listen, bind, accept, send, recv and to hadle the error thrown by these functions.

```
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unistd.h>
```

Some constant are requried to compare the Error and to declare the size of buffer

```
#define ERROR -1
#define DATA_LENGTH 512
```

To store the information about server and client socket sockaddr-in structure is used. sock and new store the server and file descriptor.

```
struct sockaddr_in server;  // Store server socket information
struct sockaddr_in client; // Store client socket information
```

sock and new variable are used to store file descriptor for server and client respectively. sockaddr-len variable is used to store the structure length client socket.

```
int sock;                // Server socket descriptor
int new;                 // Client socket descriptor
socklen_t sockaddr_len;
sockaddr_len = sizeof(struct sockaddr_in);
int data_len;
char data[DATA_LENGTH];
```

socket method is used to create socket. It takes three arguments, address type, transmission type and protocol. AF-INET is for IPv4 addressing. Socket method return a file descriptor else -1. After creating socket assign the values to the server socket. Which family it belong, port number, listening interface

```
if((sock = socket(AF_INET, SOCK_STREAM, 0)) == ERROR)
{
   perror("Server socket : ");
   exit(-1);
}
server.sin_family = AF_INET;           // (AF_INET -> ipv4)
server.sin_port = htons(atoi(argv[1]));    // Assigning port
server.sin_addr.s_addr = INADDR_ANY;       // Interface
bzero(&server.sin_zero, 8);
```

Bind method is used to bind the sock to a file descriptor, It takes socket, socket length and an integer variable. It stores the file descriptor in specified integer variable. It returns -1 if error occur. Listen method is used to put the sock on listening mode. It take file descriptor and number of client can connect to this port. It will return -1 if someting went worng else positive value.

```
if((bind(sock, (struct sockaddr *)&server, sockaddr_len)) ==
    ERROR)
{
```

```
    perror("Bind : ");
    exit(-1);
  }
  if((listen(sock, 1)) == ERROR)              // Putting socket on
      listening mode
  {
    perror("Listen :");
    exit(-1);
  }
```

accept method accepts the first pending connection in the queue. It fills the client socket structure with the details inside the packet. It returns a file descriptor else -1.
recv method is called to get the data from client file descriptor (new). recv method fills the buffer with data that is there in specified file descriptor. It takes four arguments, file descriptor, void pointer (buffer), size of buffer. It returns the number of bytes received else -1 (error).
After reading all the data send method is used to send the exact same to client. It writes the buffer data into the file descriptor.

```
  while(1)
  {
    new = accept(sock, (struct sockaddr *)&client,
        &sockaddr_len); // Accepts the first pending connection
        in the queue
    if(new == ERROR)              // Accept method also set the
        address and port number-
    {                      // to client socket descriptor
      perror("accept :");
      exit(-1);
    }

    printf("New client connected from port no %d and IP %s \n",
        ntohs(client.sin_port), inet_ntoa(client.sin_addr));
    data_len = 1;
    while(data_len)
    {
      data_len = recv(new, data, DATA_LENGTH, 0);  // Reading
          the data from client socket descriptor
```

```
        if(data_len)
        {                  // new is client socket descriptor
           send(new, data, data_len, 0);    // Send the same data
              back to client
           data[data_len] = '\0';
           printf("Sent message : %s", data);
        }
     }
     printf("Client Disconnected.\n");
     close(new);                // Closing client file descriptor
  }
  close(sock);                    // Closing server file
     descriptor

}
```

2.3 Client Implementation

A client of an Echo server suppose to send a string to server and then server will reply back with same string.

```
#include<stdio.h>
#include<stdlib.h>

// To use basic networking functions.
#include<sys/socket.h>

// To get the cause of an error
#include<errno.h>

// To perform operations on string like:- strlen
#include<string.h>

// To perform conversion of data like
#include<arpa/inet.h>
#include<unistd.h>

#define ERROR -1
#define BUFFER 512
```

The initail two step client side is as same as server side defining socket structure, creating socket structure. Once socket has been created connect method is called to connect to server. Once connection is established, send method used to send the data to server. close method is used to close the file discriptor.

```
int main(int argc, char **argv)
{
   struct sockaddr_in server; // Store server socket information
   int sock;        // Server socket descriptor
   char input[BUFFER];
   char output[BUFFER];
   int len;
   if((sock = socket(AF_INET, SOCK_STREAM, 0)) == ERROR)
   {
      perror("Socket :");
      exit(-1);
   }
   server.sin_family = AF_INET;
   server.sin_port = htons(atoi(argv[2]));
   server.sin_addr.s_addr = inet_addr(argv[1]);
   bzero(&server.sin_zero, 8);
   if((connect(sock, (struct sockaddr *)&server, sizeof(struct
       sockaddr_in))) == ERROR)
   {
      perror("Connect :");
      exit(-1);
   }
   while(1)
   {
      printf("\nEnter message : ");
      fgets(input, BUFFER, stdin);  // Take input stirng from user
      send(sock, input, strlen(input), 0);
      len = recv(sock, output, BUFFER, 0);
      output[len] = '\0';
      printf("\nReply from Srever : %s", output);
   }

   close(sock);       // Close the server descriptor
}
```

3 Application 2 : File Transfer

3.1 Objective

The objective of this program is make client server such that they can send or receive files from each other. Along with file transfer, client can also run some command on remote server to get the information about files stored on server. For example current working directory (pwd), change directory(cd), list of file (ls).

3.2 Server Implementation

At basic socket programming is done by, Creating a socket, binding, putting socket on listning mode and accepting the connections from client. These funcationalities are same echo server except the incoming request, requested by the client. Code to for basic socket programming is given below.

```
#include <stdio.h>
#include <stdlib.h>

// To avail Basic networking functionality
#include <sys/socket.h>
#include<arpa/inet.h>
#include <string.h>

// To get the cause of an error
#include<errno.h>

// To close the socket descriptor
#include <unistd.h>


// To get the file size using stat function
#include<sys/stat.h>

// To write the content of one file descriptor to another, using
    sendfile()
#include<sys/sendfile.h>

// To specify file access like:- O_RDONLY (open in read only mode)
```

```
#include<fcntl.h>

#define ERROR -1

int main(int argc,char *argv[])
{
   struct sockaddr_in server; // Store server socket information
   struct sockaddr_in client; // Store client socket information

   int serverFileDescriptor, clientFileDescriptor;
   char buf[100], command[5], filename[20];
   int k, i, size, sockaddr_len, c;
   int filehandle;
   struct stat fileInfo;
   if((serverFileDescriptor = socket(AF_INET, SOCK_STREAM, 0)) ==
      ERROR)
   {
      perror("Socket :");
      exit(1);
   }
   server.sin_family = AF_INET;
   server.sin_port = atoi(argv[1]);
   server.sin_addr.s_addr = INADDR_ANY;
    bzero(&server.sin_zero, 8);
   if(bind(serverFileDescriptor, (struct sockaddr*)&server,
      sizeof(server)) == ERROR)
   {
      perror("Bind :");
      exit(1);
   }
   if(listen(serverFileDescriptor, 1) == ERROR) // Put the server
      on listening mode
   {
      perror("Listen :");
      exit(1);
   }
   sockaddr_len = sizeof(client);     // Accepts the first pending
      connection in the queue
   clientFileDescriptor = accept(serverFileDescriptor, (struct
      sockaddr*)&client, &sockaddr_len);
```

After accepting the connection, server extract the content file descriptor to buffer (buf). The foremost string of client request contains the operation code (get, put, ls, pwd, cd, quit). get is to get the file from server, put is send a file to server, ls is to get list files available in server's current working directory, pwd current working directory, cd change directory and quit to disconnect. Now we parse the buffer and compare it every code and hadle the request accordingly.

1) Handling 'ls' request

system call is use execute ls command on server. It will store all files available in current working directory in temps.txt file. stat function is used to get the info about file tmeps.txt. First server sends size of file to client using send function. Now open function is used to open file in read only mode (O-RDONLY) it will return a file descriptor which is stored in fileHandle variable. At last sendfile function is used to send requested file. sedfile copy the content of one file descriptor to another file descriptor. It takes four arguments, dest FD, source FD, offset, size.

```
while(1)
{
   recv(clientFileDescriptor, buf, 100, 0);
   sscanf(buf, "%s", command);
   if(!strcmp(command, "ls"))
   {
      system("ls >temps.txt");
      stat("temps.txt", &fileInfo);
      size = fileInfo.st_size;
      send(clientFileDescriptor, &size, sizeof(int), 0);
      filehandle = open("temps.txt", O-RDONLY);
      sendfile(clientFileDescriptor, filehandle, NULL, size);
   }
```

2) Handling 'get' request

While parsing client request server extract file name and extension and store it on filename variable. stat function is used to get the about the file. open is called to open the file in read only mode. If file is not present then

it will return -1 else FD. If file is not present server send 0 to client else size of the file. If requested file present in server, server send that file to client using sendfile method. send file copy the content of one FD to another FD.

```
else if(!strcmp(command,"get"))
{
   sscanf(buf, "%s%s", filename, filename);
   stat(filename, &fileInfo);
   filehandle = open(filename, O_RDONLY);
   size = fileInfo.st_size;
   if(filehandle == -1)
   {
      size = 0;
   }
   send(clientFileDescriptor, &size, sizeof(int), 0);
   if(size)
   {
      sendfile(clientFileDescriptor, filehandle, NULL, size);
   }

}
```

3) Handling 'put' request

While parsing client request server extract file name and extension and store it on filename variable. Server also receives file size from client. It stored in size variable. Now server checks that file is already there or not. If file is with same name is there in current working directory is change the file name for new file by adding 1 in extension part. Once file name decided recv function is called to read the file from client FD copy to the buffer (f). Finally through write function server write buffer data in file and send positive value to client (number of bytes written).

```
else if(!strcmp(command, "put"))
{
   int c = 0, len;
   char *f;
   sscanf(buf+strlen(command), "%s", filename);
```

```
   recv(clientFileDescriptor, &size, sizeof(int), 0);
   i = 1;
   while(1)
   {
      filehandle = open(filename, O_CREAT | O_EXCL | O_WRONLY,
          0666);
      if(filehandle == -1)
      {
         sprintf(filename + strlen(filename), "%d", i);
      }
      else
      {
         break;
      }
   }
   f = malloc(size);
   recv(clientFileDescriptor, f, size, 0);
   c = write(filehandle, f, size);
   close(filehandle);
   send(clientFileDescriptor, &c, sizeof(int), 0);
}
```

4) Handling 'pwd' request

system call is used to get the current working directory of server. It will write the path in the temp.txt file. Now sever read the content of this file and sends to client. fopen function is used to open a file. It takes two arguments file name, mode.

```
else if(!strcmp(command, "pwd"))
{
   system("pwd>temp.txt");
   i = 0;
   FILE *f = fopen("temp.txt","r");
   while(!feof(f))
   {
      buf[i++] = fgetc(f);
   }
```

```
   buf[i-1] = '\0';
   fclose(f);
   send(clientFileDescriptor, buf, 100, 0);
}
```

5) Handling 'cd' request

By parsing requested string, server extracts the path. Once path has been extracted server call chdir to change the directory. If requested directory is correct it returns 0 else 1.

```
else if(!strcmp(command, "cd"))
{
   if(chdir(buf+3) == 0)
   {
      c = 1;
   }
   else
   {
          c = 0;
   }
   send(clientFileDescriptor, &c, sizeof(int), 0);
}
```

6) Handling 'quit' request

Quit code send by client when client wants to disconnet. After receiving quit server deallocate all the resources to that client.

```
else if(!strcmp(command, "quit"))
{
   printf("FTP server quitting..\n");
   i = 1;
   send(clientFileDescriptor, &i, sizeof(int), 0);
   close(serverFileDescriptor);
   close(clientFileDescriptor);
   exit(0);
   }
```

```
}
return 0;
}
```

3.3 Client Implementation

Client can send put, get, pwd, cd, ls to server. This codes are processed at server side as described above and appropriate reply sends by server.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h
#include <sys/sendfile.h>
#include <fcntl.h>

#define ERROR -1

int main(int argc,char *argv[])
{
   struct sockaddr_in server;
   struct stat obj;
   int serverFileDescriptor;
   int choice;
   char buf[100], command[5], filename[20], *f;
   int size, status;
   int filehandle;
   if((serverFileDescriptor = socket(AF_INET, SOCK_STREAM, 0)) ==
      ERROR)
   {
      perror("socket :");
      exit(1);
   }
```

```
server.sin_family = AF_INET;
server.sin_port = atoi(argv[1]);
server.sin_addr.s_addr = 0;
if(connect(serverFileDescriptor, (struct sockaddr*)&server,
    sizeof(server)) == ERROR)
{
   perror("Connect :");
   exit(1);
}
int i = 1;
while(1)
{
   printf("\nEnter 1 to get the file from server:");
   printf("\nEnter 2 to put the file to server:");
   printf("\nEnter 3 to get the current working directory of
       server:");
   printf("\nEnter 4 to list the file available :");
   printf("\nEnter 5 to change the directory :");
   printf("\nEnter 6 to Quit:");
   printf("\nEnter a choice :");
   scanf("%d", &choice);
   switch(choice)
   {
```

Once client establishes a connection with server it can send any of six request to server.

1) Requesting 'get'

Client program takes the file name from user and save it in filename variable. Fill the buffer with 'get ' string and file name. These two parameters together form a get request. Client send this request to server and get the file size as non zero if file exsits in server if not 0. If file size non-zero recv function is called to get the file.

```
case 1:
printf("\nEnter filename : ");
scanf("%s", filename);
strcpy(buf, "get ");
strcat(buf, filename);
```

```
send(serverFileDescriptor, buf, 100, 0);
recv(serverFileDescriptor, &size, sizeof(int), 0);
if(!size)
{
   printf("\nFile doesn't exists.\n");
   break;
}
f = malloc(size);
recv(serverFileDescriptor, f, size, 0);
while(1)
{
   filehandle = open(filename, O_CREAT | O_EXCL | O_WRONLY, 0666);
   if(filehandle == -1)
   {
      sprintf(filename + strlen(filename), "%d", i);
   }
   else
   {
      break;
   }
   write(filehandle, f, size, 0);
   close(filehandle);
   strcpy(buf, "cat ");
   strcat(buf, filename);
   system(buf);
   break;
```

2) Requesting 'put'

Client program takes the file name from user and open it using open method. If file is not available open returns -1 else FD. Now client insert 'put ' string inside buffer (buf) and the file name and send it to server. After that client send the size of the file. At last client send file using sendfile method.

```
case 2:
   printf("\nEnter filename : ");
   scanf("%s", filename);
   filehandle = open(filename, O_RDONLY);
```

```
if(filehandle == -1)
{
   printf("File doesn't exists\n\n");
   break;
}
strcpy(buf, "put ");
strcat(buf, filename);
send(serverFileDescriptor, buf, 100, 0);
stat(filename, &obj);
size = obj.st_size;
send(serverFileDescriptor, &size, sizeof(int), 0);
sendfile(serverFileDescriptor, filehandle, NULL, size);
recv(serverFileDescriptor, &status, sizeof(int), 0);
if(status)
{
   printf("File stored\n");
}
else
{
   printf("\nFile transmission failed");
}
break;
```

3) Requesting 'PWD'

Client copy 'pwd' string in buffer(buf) using strcpy function and send it to server. Server will reply with the current directory of server. Finally client get that path using recv method and print it.

```
case 3:
   strcpy(buf, "pwd");
   send(serverFileDescriptor, buf, 100, 0);
   recv(serverFileDescriptor, buf, 100, 0);
   printf("The path of the remote directory is: %s\n", buf);
   break;
```

4) Requesting 'ls'

Client will simply send 'ls' string to server and server send the list of files. Client stores all the file names in temp.txt and print it.

```
case 4:
   strcpy(buf, "ls");
   send(serverFileDescriptor, buf, 100, 0);
   recv(serverFileDescriptor, &size, sizeof(int), 0);
   f = malloc(size);
   recv(serverFileDescriptor, f, size, 0);
   filehandle = open("temp.txt", O_CREAT|O_WRONLY, 0666);
   write(filehandle, f, size, 0);
   printf("List of files at remote directory:\n");
   system("cat temp.txt");
   close(filehandle);
   break;
```

5) Requesting 'cd'

Client take path form user and send to server after concate this with 'cd ' string. If server reply with 1 means path changed else path doesn't exsits.

```
case 5:
   strcpy(buf, "cd ");
   printf("Enter the path to change the remote directory: ");
   scanf("%s", buf + 3);
   send(serverFileDescriptor, buf, 100, 0);
   recv(serverFileDescriptor, &status, sizeof(int), 0);
   if(status)
   {
      printf("Remote directory successfully changed\n");
   }
   else
   {
      printf("Remote directory failed to change\n");
   }
   break;
```

6) Requesting 'quit'

Quit command disconnect the connection between server and client.

```
case 6:
   strcpy(buf, "quit");
   send(serverFileDescriptor, buf, 100, 0);
   recv(serverFileDescriptor, &status, 100, 0);
   if(status)
   {
      printf("Server closed\nQuitting..\n");
      exit(0);
   }
   printf("Server failed to close connection\n");
   close(serverFileDescriptor);
   }
}
}
```

4 Application 3 : HTTP Server

A HTTP Server also known as web server responds to requests from web browsers like firefox or chrome. It makes decisions about what content (html, css, js) from disk should be sent to the client. Often merging that static content with dynamic content from a database or service.

4.1 Obective

To build a HTTP Server capable of handling get, put request from any HTTP Client (e.g. web browser).

4.2 Implementation

The basic desing of HTTP server is more or less similar to our file upload server. But communication between client and server is done in different manner. The way client request for a resource from server is quit standardized. The HTTP Server implemented in this is a very tiny version of real HTTP Server. This HTTP server can handle only get or put request.
The implementation of HTTP Server is divided into three parts.

- Main : Accepts the client requests and manage it.
- Start Server : Keeps the server process on
- Respond : Servers the client request

1) Main
This part of HTTP server contains basic functionality of socket programming. List of constants are required to track maximum clients, buffer size, error, path. List of global variable are used to track server FD, array of client FDs. The main responsibility is accept the clients request and create the corresponding client and store it on client array. This array tracked using slot variable. This main function forked to create a child process to handle a client and simultaneously accepting connections to other clients.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/sendfile.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>

#define CONNMAX 1000
#define BYTES 1024
#define ERROR -1
char *ROOT;
int listenfd, clients[CONNMAX];
void startServer(char *);
void respond(int);

int main(int argc, char* argv[])
{
   struct sockaddr_in clientaddr;
   socklen_t addrlen;
   char c;
   int slot;
   int i;
   char PORT[6];
   ROOT = getenv("PWD");
   strcpy(PORT,"10000");
   for (i = 0; i < CONNMAX; i++)
   {
      clients[i] = -1;
   }
   startServer(PORT);

   while (1)
   {
   addrlen = sizeof(clientaddr);

   // Accept connection
```

```
   clients[slot] = accept(listenfd, (struct sockaddr *)
       &clientaddr, &addrlen);

   if (clients[slot] < 0)
   {
      perror ("accept() error");
   }
   else
   {
      // Create a child process to responed to client
      if (fork() == 0)
      {
         // Handle the client at slot
         respond(slot);
         exit(0);
      }
     }
    while (clients[slot] != -1)
   {
      slot = (slot+1) % CONNMAX;
   }
    }

    return 0;
}
```

---

2) Start Server

This function create server socket and bind the server FD with the server socket. finally puts the server on listening mode to accept the connection.

```
void startServer(char *port)
{
   struct addrinfo hints, *res, *p;

   // getaddrinfo for host
   memset (&hints, 0, sizeof(hints));
```

```
   hints.ai_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE;

   if(getaddrinfo( NULL, port, &hints, &res) != 0)
   {
      perror ("getaddrinfo() error");
      exit(1);
   }

   // socket and bind
   for(p = res; p!=NULL; p=p->ai_next)
   {
      listenfd = socket (p->ai_family, p->ai_socktype, 0);
      if(listenfd == -1)
      {
         continue;
      }

      if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
      {
         break;
      }
      }

   if (p==NULL)
   {
      perror ("socket() or bind()");
      exit(1);
   }

   freeaddrinfo(res);

   // listen for incoming connections
   if ( listen (listenfd, 1000000) != 0 )
   {
      perror("listen() error");
      exit(1);
   }
}
```

3) Respond
This function used to parse the HTTP client request. This funcation can only parse put or get request. strtok function is used to generate the tokens from the request. This server first check 4 byte of request if that contains either get or put then only this will respond. Once server found that request is correct then it will first send an OK message to client ('HTTP/1.0 200 OK/n/n'). After that it send the requested file to client. If client doesn't specify any resource then server returns index.hmtl by default.

```
void respond(int n)
{
   char mesg[99999], *reqline[3], data_to_send[BYTES], path[99999];
   int rcvd, fd, bytes_read;
   char *token, *temp;
   const char s[2] = "=";
   // mesg buffer filled with '\0'
   memset((void*)mesg, (int)'\0', 99999);
   // Copy the content of socket descriptor clients[n] to mesg
       buffer
   rcvd=recv(clients[n], mesg, 99999, 0);
   // Error checking
   if (rcvd < 0)
   {
      fprintf(stderr, ("recv() error\n"));
   }
   else if (rcvd == 0)
   {
      fprintf(stderr, "Client disconnected upexpectedly.\n");
   }
   else
   {
      printf("%s", mesg);
      // Breaks the string into tokens
      reqline[0] = strtok (mesg, " \t\n");

      // Handle GET request
      if ( strncmp(reqline[0], "GET\0", 4)==0 )
```

```
{
   reqline[1] = strtok (NULL, " \t");
   reqline[2] = strtok (NULL, " \t\n");
   if (strncmp( reqline[2], "HTTP/1.0", 8)!=0 && strncmp(
       reqline[2], "HTTP/1.1", 8)!=0)
   {
      write(clients[n], "HTTP/1.0 400 Bad Request\n", 25);
   }
   else
   {
      if (strncmp(reqline[1], "/\0", 2) == 0)
      {
      reqline[1] = "/index.html";
      }
      strcpy(path, ROOT);
      strcpy(&path[strlen(ROOT)], reqline[1]);
      printf("file: %s\n", path);

      if ((fd=open(path, O_RDONLY)) != -1)
      {
         send(clients[n], "HTTP/1.0 200 OK\n\n", 17, 0);
         while ((bytes_read=read(fd, data_to_send, BYTES))>0)
         write (clients[n], data_to_send, bytes_read);
      }
      else
      {
         //File not found
         write(clients[n], "HTTP/1.0 404 Not Found\n", 23);
      }
           }
      }
// handle PUT request
else if(strncmp(reqline[0], "PUT\0", 4) == 0)
{
   reqline[1] = strtok (NULL, " \t");
   reqline[2] = strtok (NULL, " \t\n");
   if (strncmp( reqline[2], "HTTP/1.0", 8)!=0 && strncmp(
       reqline[2], "HTTP/1.1", 8) != 0)
   {
      write(clients[n], "HTTP/1.0 400 Bad Request\n", 25);
```

```
        }
        else
        {
           while(token != NULL)
           {
              strcpy(temp, token);
              token = strtok(NULL, s);
           }
           printf("Data accepted : %s", "Have a nice day!");
           write(clients[n], "Data accepted\n", 14);
                }
            }
     }

   //Closing SOCKET : this makes this server stateless
   shutdown (clients[n], SHUT_RDWR);
   close(clients[n]);
   clients[n]=-1;
}
```

Note :- It may possible that this HTTP Server might not responed to 'put' method. The reason is that all the Beta browsers like :- Chrome, Mozila etc. stopped supporting 'put' method on basic HTML page. It will work fine if HTML page using ajax and other methods to send put request..

5 Application 4 : HTTP Client

5.1 Objective

The objective is to create a program that can send get or put request to a real HTTP Server.

5.2 Implementation

To implement HTTP client the structure of 'get' and 'put' request should be known.

- GET Request format :

  GET /path/file.html HTTP/1.0
  From: someuserjmarshall.com
  User-Agent: HTTPTool/1.0
  blank line here

- PUT Request format :

  PUT /path/file.html HTTP/1.0
  From: someuserjmarshall.com
  User-Agent: HTTPTool/1.0
  Body : Data

Http client two separate buffer fro get and put request. Client create socket and connect to the server that we create in last application. It send the get or put request by using send function. corresponding response will handle by this recv function. This client doesn't suport HTML, it will just print the server's response on terminal.

```
#include <stdio.h>
#include <stdlib.h>
#include <syssocket.h>
#include <errno.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define ERROR -1
#define BUFFER 2048


int main(int argc, char **argv)
{
   char output[BUFFER];
   int len;
   struct sockaddr_in server;
```

```
   int sock;
   char input[BUFFER]="GET /index.html HTTP/1.1 \n User-Agent:
      Mozilla/4.0 (compatible; MSIE5.01; Windows   NT)\nHost:
      127.0.0.1:1000\nAccept-Language: en-us\nAccept-Encoding:
      gzip, deflate\nConnection: Keep-  Alive";

   char in[BUFFER] = "PUT /hello.htm HTTP/1.0-User=Agent:
      Mozilla/4.0 (compatible; MSIE5.01; Windows    NT)=Host:
      127.0.0.1:10000=Accept-Language: en-us=Connection:
      Keep-Alive=Content-type: text/     html=Content-Length:
      182=Have a nice day!";

   if((sock = socket(AF-INET, SOCK-STREAM, 0)) == ERROR)
   {
      perror("Socket :");
      exit(-1);
   }
   server.sin_family = AF-INET;
   server.sin_port = htons(atoi(argv[2]));
   server.sin_addr.s_addr = inet_addr(argv[1]);
   bzero(&server.sin_zero, 8);

   if((connect(sock, (struct sockaddr *)&server, sizeof(struct
       sockaddr_in))) == ERROR)
   {
      perror("Connect :");
      exit(-1);
   }

   while(1)
   {
      send(sock, input, strlen(input), 0);
      len = recv(sock, output, BUFFER, 0);
      output[len] = '\0';
      printf("%s", output);
   }

   close(sock);
}
```

A Appendix

- Implemented code is available on git rapo.

  https://github.com/sourabhjains/socket-programming-using-c