Programming Assignment I

Introduction to Socket Programming in C/C++

Computer Networks | 18/11/2018

Arsanuos Essa (17)

Amr Hendy (46)

Mohammed Deifallah (59)

Introduction

In this assignment, it's required to use sockets to implement a simple web client that communicates with a web server using a restricted subset of HTTP.

Multi-threaded Web Server

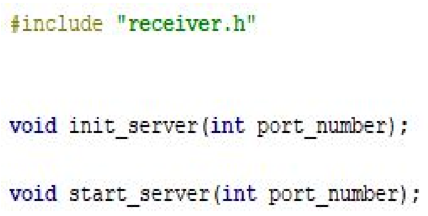
Pseudocode

```
while true: do
    Listen for connections
    Accept new connection from incoming client and delegate it to worker thread/process
    Parse HTTP/1.0 request and determine the command (GET or POST)
    Determine if target file exists (in case of GET) and return error otherwise
    Transmit contents of file (reads from the file and writes on the socket) (in case of GET)
    Close the connection
end while
```

Organization

server.h

It's a simple header file for the server side to behave as an interface for the functionalities provided by the server. As shown below, the header file contains two simple functions with one import to another utility header file.



```
#include "receiver.h"


void init_server(int port_number);

void start_server(int port_number);
```

server.cpp

It includes the implementation of the functions mentioned above. It has some system calls as `socket(address.sin_family, SOCK_STREAM, 0))` ,

`bind(server_socketfd, (struct sockaddr *)&address, sizeof(address))`

`setsockopt(server_socketfd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))` ,

`listen(server_socketfd, SERVER_CONNECTION_QUEUE_SIZE)` ,

`accept(server_socketfd, (struct sockaddr *)&client_addr, &sock_size);` .

Handling Closing Connections

To handle the persistent connections, the server will not close the client connection after serving the request and leave it open, the server will close the client connection if TIMEOUT exceeded from the last request received from that client, we handled that TIMEOUT value to be dynamic by make it inversely proportional with the number of active client connections in the server.

TIMEOUT = (MAX_CONNECTION_LIMIT / Active Clients) * MINIMUM_TIMEOUT

MAX_CONNECTION_LIMIT we assume it to be 20

MINIMUM_TIMEOUT we assume it to be 2 seconds

myserver.cpp

The to-be-called main of the server side.

POST HANDLING

When server receives POST request it sends OK response to the client after that the client starts to send file while server receiving it. In general server or client receives data in chunks and make sure to only reads the size of the sent data on more and no less.

The receiving function is built on top of socket function recv with flag MSG_PEEK that means don't remove the received data from socket. putting it inside a while loop to handle the case that we couldn't read all data.

GET HANDLING

When server receives GET it directly responds with the required file if exists and responds with 404 if the file doesn't exists.

In case of pipelining, Server receiver all GET requests from client once and responds once the it will be the responsibility of the client to receive the response in appropriate manner.

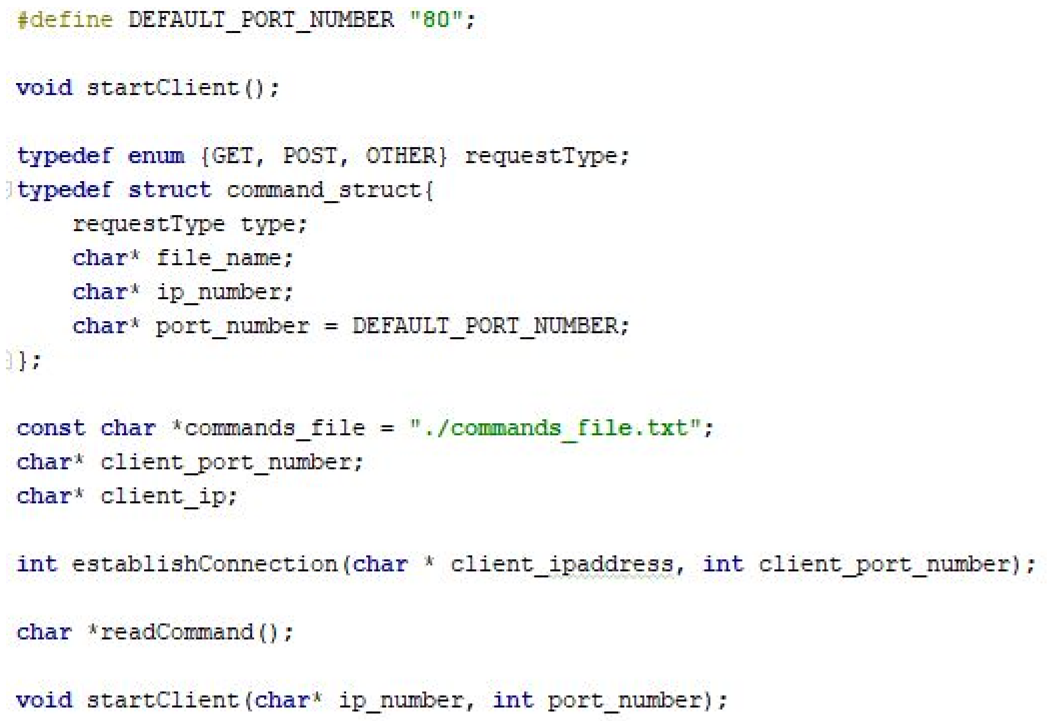
Client

Pseudocode

```
while more operation exists do
    Create a TCP connection with the server
    Wait for permission from the server
    Send next requests to the server
    Receives data from the server (in case of GET) or sends data (in case of POST)
    Close the connection
end while
```

Organization

client.h



```
#define DEFAULT_PORT_NUMBER "80";

void startClient();

typedef enum {GET, POST, OTHER} requestType;
typedef struct command_struct{
    requestType type;
    char* file_name;
    char* ip_number;
    char* port_number = DEFAULT_PORT_NUMBER;
};

const char *commands_file = "./commands_file.txt";
char* client_port_number;
char* client_ip;

int establishConnection(char * client_ipaddress, int client_port_number);

char *readCommand();

void startClient(char* ip_number, int port_number);
```

As shown above, it's a header file doing as the server header file does. It has an enumeration to categorize the requests. Also, it has a **struct** data structure to keep information about the command, read from the input file.

client.c

It contains the implementation of the previous header file, and it has some system calls too to handle socket connection, receiving, sending and closing.

my_client.c

The to-be-called main of the client side.

GET HANDLING

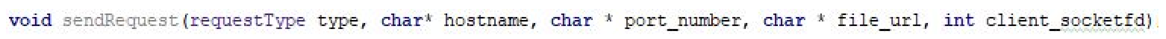
- Client first establishes a connection with the server for all the following requests.
- Client will read all the consecutive GET requests and send them all to the server not waiting for the response to send them, then it will receive all their response and write the contents into the files.
- if there is a POST request, client will send it immediately to the server then wait for the OK response to send the file to the server.
- In receiving the data content in response we receive the file on several chunks to handle large files to fit into the memory.
- Also while receiving the file we handled to read the file content only to avoid the intersection between the different data files by using MSG_PEEK flag supported in recv system call and also using Content-Length header in the response.

POST HANDLING

- Client first establishes a connection with the server for all the following requests.
- if there is a POST request, client will send it immediately to the server then wait for the OK response to send the file to the server, if the client doesn't receive from server the OK response for 5 seconds it will ignore the post request, otherwise the client will send the file data to the server.

Utilities

SENDER.H



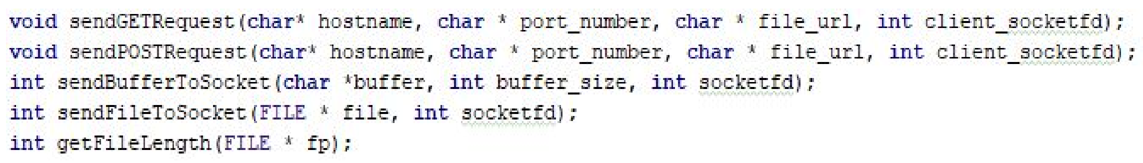
```
void sendRequest(requestType type, char* hostname, char * port_number, char * file_url, int client_socketfd)
```

It only has the function above which is responsible for sending a request from some the host/client to the server side.

This sendRequest function will redirect the request to the correct sending function according to the request type (GET/POST).

SENDER.C

It implements the mentioned function through some other utility functions. The figure below shows those functions:

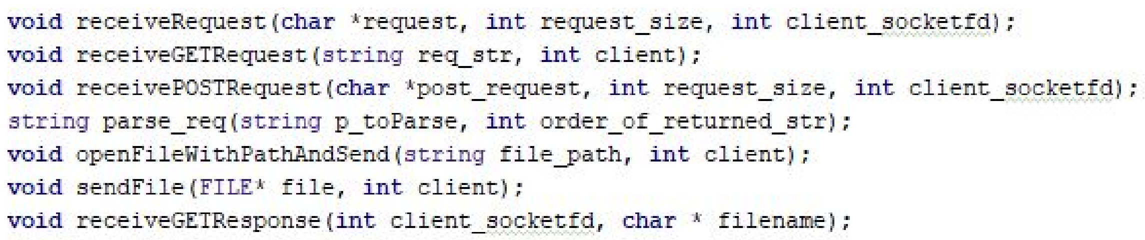


```
void sendGETRequest(char* hostname, char * port_number, char * file_url, int client_socketfd);
void sendPOSTRequest(char* hostname, char * port_number, char * file_url, int client_socketfd);
int sendBufferToSocket(char *buffer, int buffer_size, int socketfd);
int sendFileToSocket(FILE * file, int socketfd);
int getFileLength(FILE * fp);
```

There's no need to note that many file-related system calls are used in that file, such as **ftell()**, **fseek()**, and **rewind()**.

RECEIVER.H

Again, it has the interface to be implemented by any side receiving an attachment. The figure of that interface is shown below:



```
void receiveRequest(char *request, int request_size, int client_socketfd);
void receiveGETRequest(string req_str, int client);
void receivePOSTRequest(char *post_request, int request_size, int client_socketfd);
string parse_req(string p_toParse, int order_of_returned_str);
void openFileWithPathAndSend(string file_path, int client);
void sendFile(FILE* file, int client);
void receiveGETResponse(int client_socketfd, char * filename);
```

RECEIVER.C

It implements the functions mentioned before. It's worthy to say that **Regex** is used to categorize the request type.

Also to receive all the data files correctly, we use MSG_PEEK flags supported in recv system call to return the socket pointer to recv from the correct start position, this is controlled by using Content-Length header in the response.