

Academia Tehnică Militară Ferdinand I
Facultatea de Sisteme Informatice și Securitate Cibernetică



Proiect Proiectarea Sistemelor de Operare
“Mini-Server Web”

Student: Mistic Ioana

Grupa: C113-D

Cuprins

Capitolul 1. Introducere.....	3
Capitolul 2. Cerințe de sistem.....	4
Capitolul 3. Funcționalități sistem client-server	4
Capitolul 4. Descriere proiect	6
Capitolul 5. Server Linux	17
5.1. Notiuni teoretice	17
5.2. Implementare.....	19

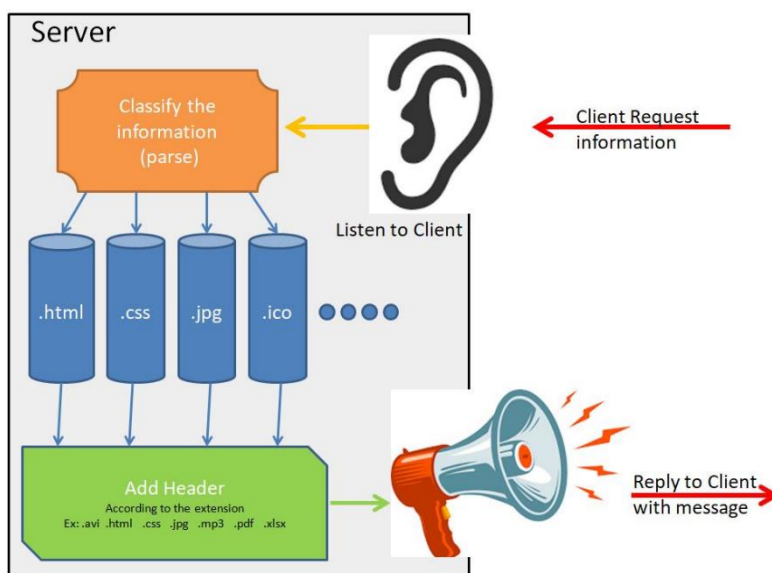
Capitolul 1. Introducere

În lumea internetului, există întotdeauna un server care poate deservi mai mulți clienți. De exemplu, Google, Netflix, Facebook și așa mai departe sunt servere. Oamenii ca noi sunt clienți și aceștia pot folosi browsere web (Chrome, Edge, Opera, Firefox, etc.) pentru a comunica cu serverele.

Serverul web sau web server este serverul care stochează (găzduiește) pagini web și le pune la dispoziția solicitanților prin protocolul HTTP. Și de dată aceasta relația server-client se bazează pe o aplicație care este instalată pe server și care este programată să transfere paginile web găzduite. Putem observa aici că ideea de web server presupune și noțiunea de hosting (găzduire), asta deoarece serverul trebuie să dețină datele pe care urmează să le returneze la cerere.

Relația este următoarea: utilizatorul (clientul) aflat în dreptul unui computer pe care are instalată o aplicație tip browser solicită (serverului) prin intermediul unui url o anumită pagină web; serverul rulează anumite linii de cod și returnează un rezultat.

Un server web acceptă și îndeplinește solicitările clienților pentru conținut static (adică, pagini HTML, fișiere, imagini și videoclipuri) de pe un site web. Serverele web gestionează numai cererile și răspunsurile HTTP .



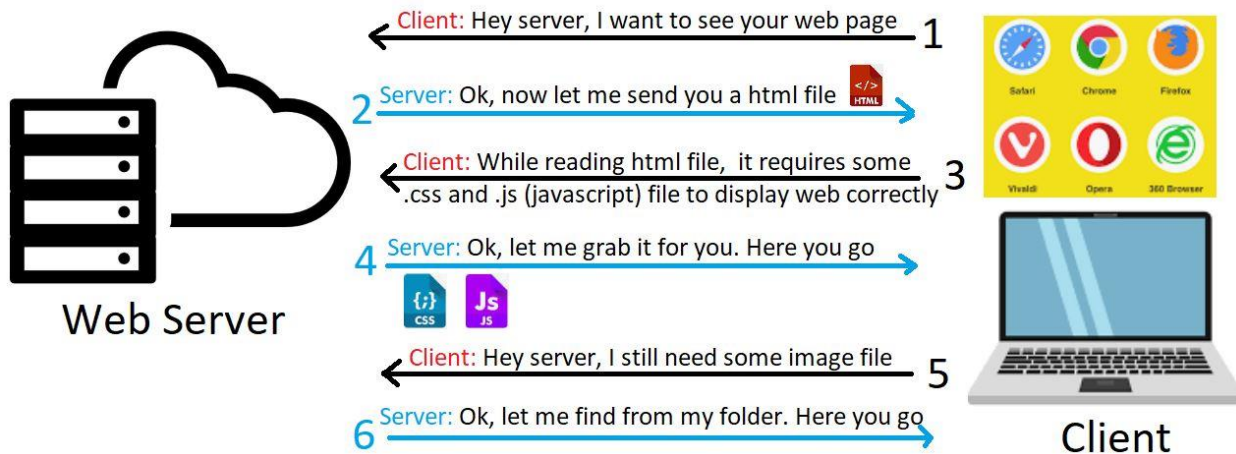
Capitolul 2. Cerințe de sistem

Rol	Cerinte
Server	Sistem de operare Linux/Windows. Mediu de dezvoltare C\C++: Visual Studio.
Client	Orice sistem de operare (Windows, IOS, Android, Ubuntu) care poate accesa browserul web.

Capitolul 3. Funcționalități sistem client-server

A.Client: clientul este utilizatorul care accesează site-ul web de pe dispozitivul său, fie el laptop, desktop, tabletă, smartphone etc. Pentru acces, este nevoie doar de o conexiune la Internet și un browser web, un software foarte important din partea clientului, care este responsabil de afișarea întregului conținut web într-un mod ușor de utilizat și de a permite utilizatorului să interacționeze cu acesta. Și pentru aceasta vom avea nevoie doar de adresa paginii web sau de un IP.

B.Server: va conține toate datele și un software care acționează ca un server, adică îi permite clientului să se conecteze pentru a face tot ce trebuie să facă. În cazul unui server web, acesta va fi, de exemplu, Apache, Lighttpd etc.



Serverele web urmează un model client-server. În această structură, un program, cunoscut și sub numele de client, solicită o resursă sau serviciu de la un alt program, server.

Pentru a procesa cererile clienților web, serverele web urmează câțiva pași:

- Când un utilizator web dorește să încarce conținutul unui site web, browserul său web solicită acces prin internet. Aceasta se numește cerere HTTP. Browserul web caută adresa IP a site-ului web solicitat traducând adresa URL a paginilor web prin sistemul de nume de domeniu (DNS) sau căutând prin cache-ul acestuia. Acest proces localizează serverul web unde sunt găzduite fișierele site-ului.
- Serverul web primește cererea HTTP și o procesează prin serverul său HTTP. Odată ce serverul său HTTP acceptă cererea, va căuta prin fișierele serverului pentru a obține datele relevante.
- După aceea, serverul web returnează fișierele site-ului browserului web care a trimis solicitarea. Apoi, utilizatorul web vede conținutul site-ului.

Cu toate acestea, dacă serverul HTTP nu reușește să găsească sau să proceseze fișierele solicitate, acesta răspunde browserului web cu un mesaj de eroare. Una dintre cele mai frecvente este o eroare 404, dar poate apărea și o eroare 403 dacă există probleme de permisiuni.

Pe de altă parte, dacă un server web nu primește un răspuns în timp util de la un alt server care acționează ca proxy sau gateway, apare o eroare 504.

Capitolul 4. Descriere proiect

Acest proiect arată una dintre modalitățile majore în care sunt implementate serverele HTTP, și anume, folosește un socket TCP pentru a asculta cererile primite și trimite înapoi un răspuns HTTP de bază bazat pe standardele stabilite de documentele RFC.

Serverul continuă să asculte orice mesaj primit, apoi trebuie să analizăm care sunt informațiile utile din mesaj, parsându-l. Informațiile utile la care ne pasă sunt numele fișierului (cu calea) și extensia fișierului. Serverul deschide apoi fișierul conform căii și pune conținutul fișierului într-un mesaj de răspuns pe care îl vom trimite ulterior clientului. Înainte de a trimite mesajul de răspuns, ar trebui mai întâi să spunem clientului ce tip de conținut de fișier vom trimite, poate fișier imagine (.jpg, .png, ...) sau fișier txt (.html, .doc, ...) și așa mai departe.

Mediul de dezvoltare al aplicației este Visual Studio. Serverele web urmează un model client-server. În această structură, un program, cunoscut și sub numele de client, solicită o resursă sau serviciu de la un alt program, server.

Pentru a comunica între software-ul de rețea Windows și serviciile de rețea, voi folosi fișierul header: `#include <WS2tcpip.h>`. Fișierul antet `Ws2tcpip.h` conține definiții introduse în documentul WinSock 2 Protocol-Specific pentru TCP/IP, care include funcții și structuri mai noi utilizate pentru a prelua adrese IP. (Winsock permite programelor și aplicațiilor Windows să se conecteze la internet prin TCP/IP).

Serverul va avea un socket TCP care:

1. ascultă cererile primite pe o anumită adresă de socket (care este în mare parte o combinație între o adresă IP și un port de rețea).
2. procesează sincron conexiunile de rețea dintr-o coadă de fire de execuție (din ce a ascultat) (acceptă o conexiune de rețea din coadă pe rând).
3. citește mesajul trimis de un client prin conexiunea la rețea și trimite un răspuns către client prin conexiunea la rețea.

1. ascultă cererile primite pe o anumită adresă de socket

Voi avea o clasa TCPListen ce va conține:

- un constructor ce va primi ca parametri adresa ip și portul

```
TCPListen(const char* ipAddress, int port) :  
    adresaIP(ipAddress), portul(port) { }
```

- funcția init():
 - creare socket

```
socketul = socket(AF_INET, SOCK_STREAM, 0);  
if (socketul == INVALID_SOCKET)  
{  
    return WSAGetLastError();  
}
```
 - Structura „sockaddr_in” este foarte frecvent utilizată în programarea socketului în limbajul de programare C. Această structură vă permite să legați un socket cu adresa dorită, astfel încât un server să poată asculta cererile de conectare ale clienților.

```
sockaddr_in adr;  
adr.sin_family = AF_INET;  
adr.sin_port = htons(portul);
```

```
inet_pton(AF_INET, adresaIP, &adr.sin_addr);
```

```
if (bind(socketul, (sockaddr*)&adr, sizeof(adr)) ==  
SOCKET_ERROR)  
{  
    return WSAGetLastError();  
}
```

- listen pentru solicitarile de conectare primite pe socketul creat
if (listen(socketul, SOMAXCONN) == SOCKET_ERROR)
{
 return WSAGetLastError();
}
- creare file descriptor, initializat cu 0, adaugam socketul de
ascultare pentru a putea intercepta conexiunea
FD_ZERO(&filedescriptorul);
FD_SET(socketul, &filedescriptorul);

- functia run():

- se va crea o copie a setului de file descriptori ai fisierului
principal. Copia contine socket-urile care accepta cereri de
conexiune de intrare sau mesaje. De exemplu, avem un server
si setul de descriptori al fisierului principal, care contine 5 item-
uri: socket-ul de ascultare si patru clienti. Cand se ajunge in
select(), sunt returnate doar socket-urile care interactioneaza
cu serverul.

```
bool running = true;
```

```
while (running)  
{
```



```

fd_set copie = filedescriptorul;
int socketCount = select(0, &copie, nullptr, nullptr, nullptr);
for (int i = 0; i < socketCount; i++)
{
    SOCKET sock = copie.fd_array[i];

    if (sock == socketul)
    {

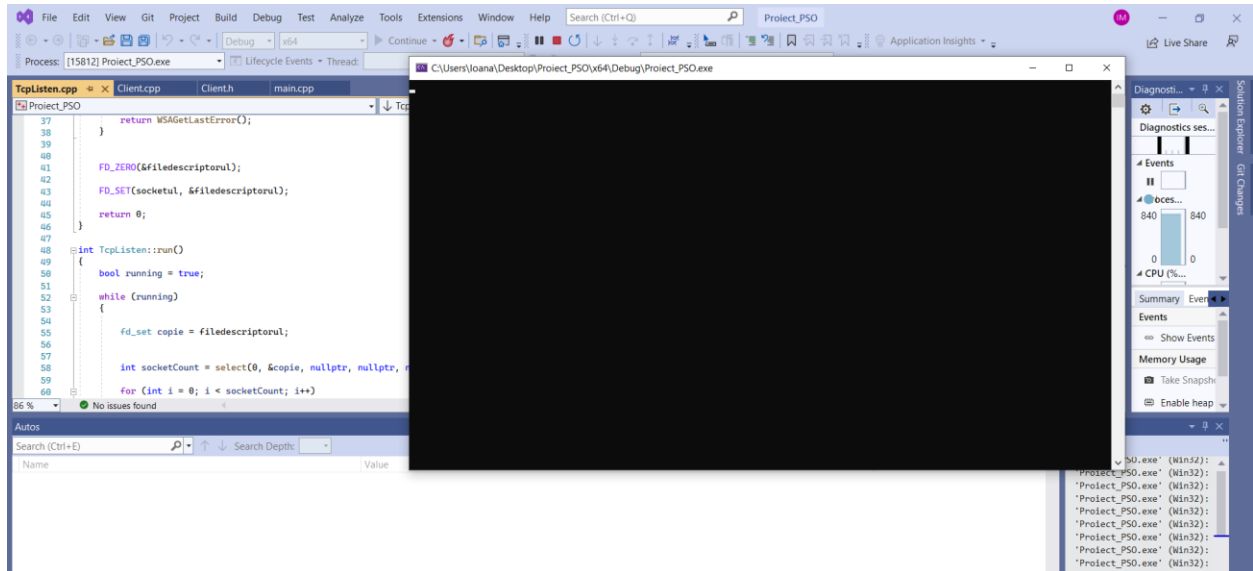
        SOCKET client = accept(socketul, nullptr, nullptr);
        FD_SET(client, &filedescriptorul);
        onClientConnected(client);
    }
    else
    {
        char buf[4096];
        ZeroMemory(buf, 4096);

        int bytesIn = recv(sock, buf, 4096, 0);
        if (bytesIn <= 0)
        {
            onClientDisconnected(sock);
            closesocket(sock);
            FD_CLR(sock, &filedescriptorul);
        }
        else
        {
            onMessageReceived(sock, buf, bytesIn);
        }
    }
}
}

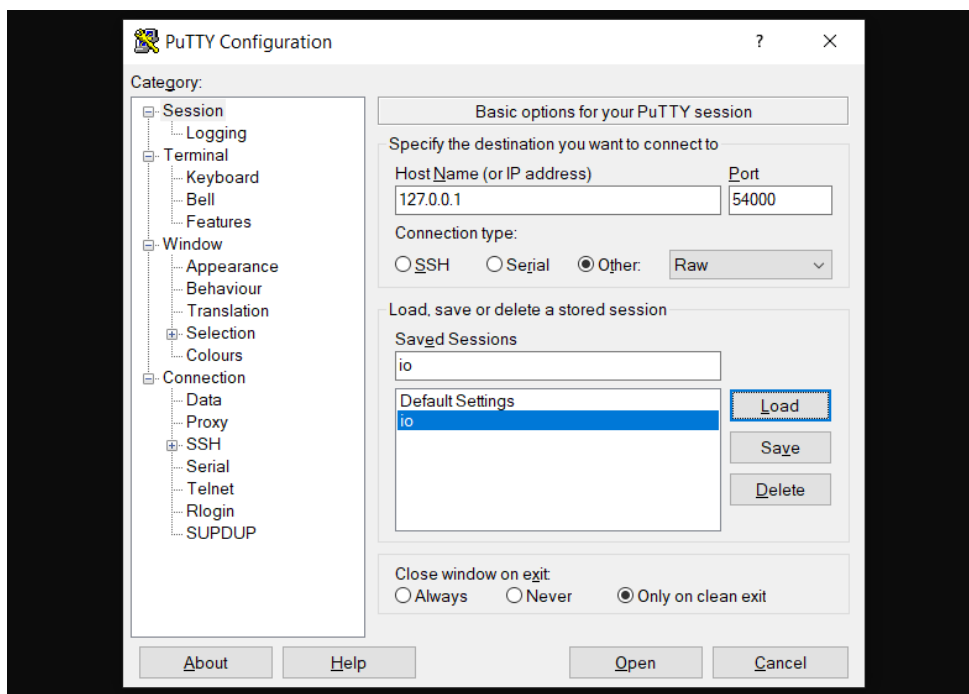
```

Rulare program:

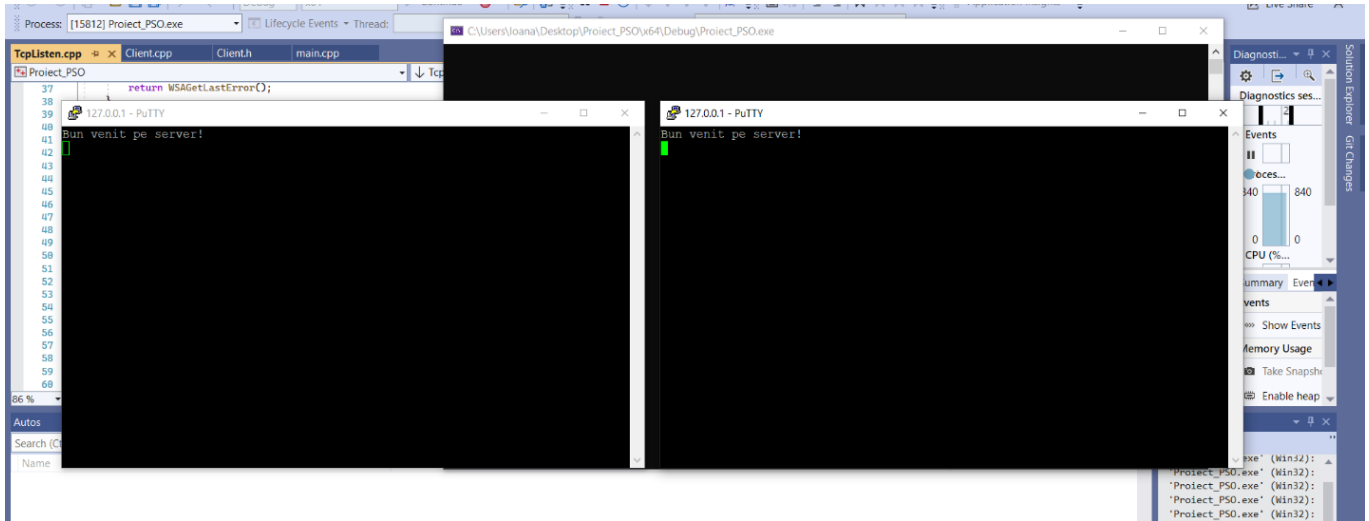
- Pornire Serverul din Visual Studio:



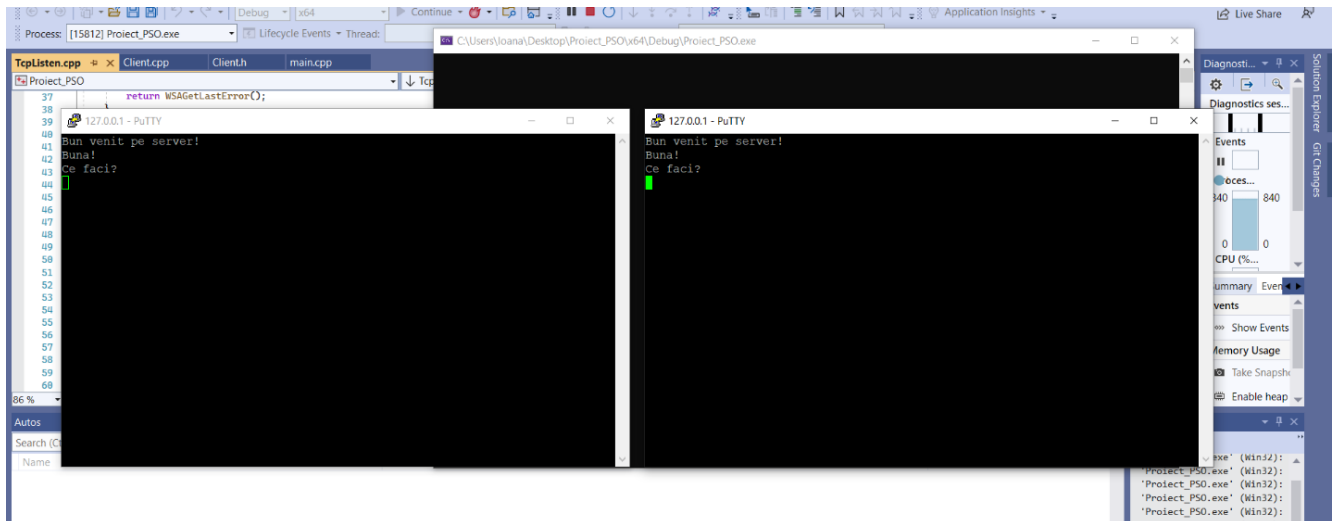
- Conectare Putty:



- Funcționalitate server (mesaje între clienți cu ajutorul Putty):



- Schimb mesaje clienți: (terminalele comunică între ele)



- Funcționalitate server web:

Voi avea o clasa WebServer prin care ne vom putea conecta la paginile web disponibile, ce va conține:

```
void WebServer::onMessageReceived(int clientSocket, const char* msg, int length)
{
    // Analizeaza sirul de solicitare al clientului, de ex. GET /index.html
    HTTP/1.1
    istreamstring iss(msg);
    vector<string> parsed((istream_iterator<string>(iss)),
    istream_iterator<string>());

    // Unele valori implicite pentru iesire catre client (404 file not found)
    string content = "<h1>404 Not Found</h1>";
    string htmlFile = "/index.html";
    int errorCode = 404;

    // Daca solicitarea GET este valida se incerca sa se obtina numele
    if (parsed.size() >= 3 && parsed[0] == "GET")
    {
        htmlFile = parsed[1];

        //if-ul de aici e in caz ca http://localhost:8080/
        if (htmlFile == "/")
        {
            htmlFile = "/index.html";
        }
    }

    ///// Deschide documentul din fisierul pagini
    ifstream f(".\\pagini" + htmlFile);

    //Verifica daca s-a deschis si daca a facut-o se ia intregul continut
    if (f.good())
    {
        string str((istreambuf_iterator<char>(f)),
    istreambuf_iterator<char>());
        content = str;
        errorCode = 200;
    }

    f.close();

    // Scrie documentul inapoi catre client
    std::ostringstream oss;
    oss << "HTTP/1.1 " << errorCode << " OK\r\n";
    oss << "Cache-Control: no-cache, private\r\n";
    oss << "Content-Type: text/html\r\n";
    //oss << "Content-Type: text/html\r\n";
    oss << "Content-Length: " << content.size() << "\r\n";
    oss << "\r\n";
    oss << content;
```

```

std::string output = oss.str();
int size = output.size() + 1;

sendToClient(clientSocket, output.c_str(), size);
}

```

În main.cpp vom crea o variabilă de tipul WebServer, unde vom avea nevoie de o adresă IP și de un port.

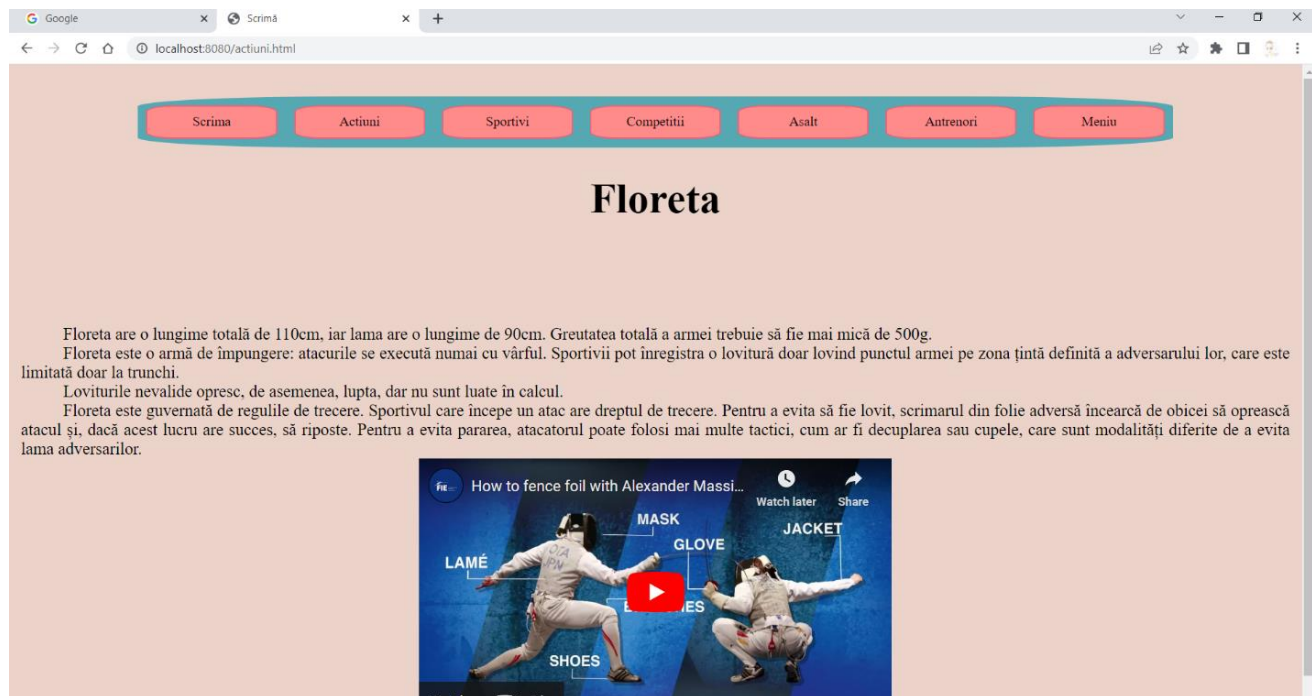
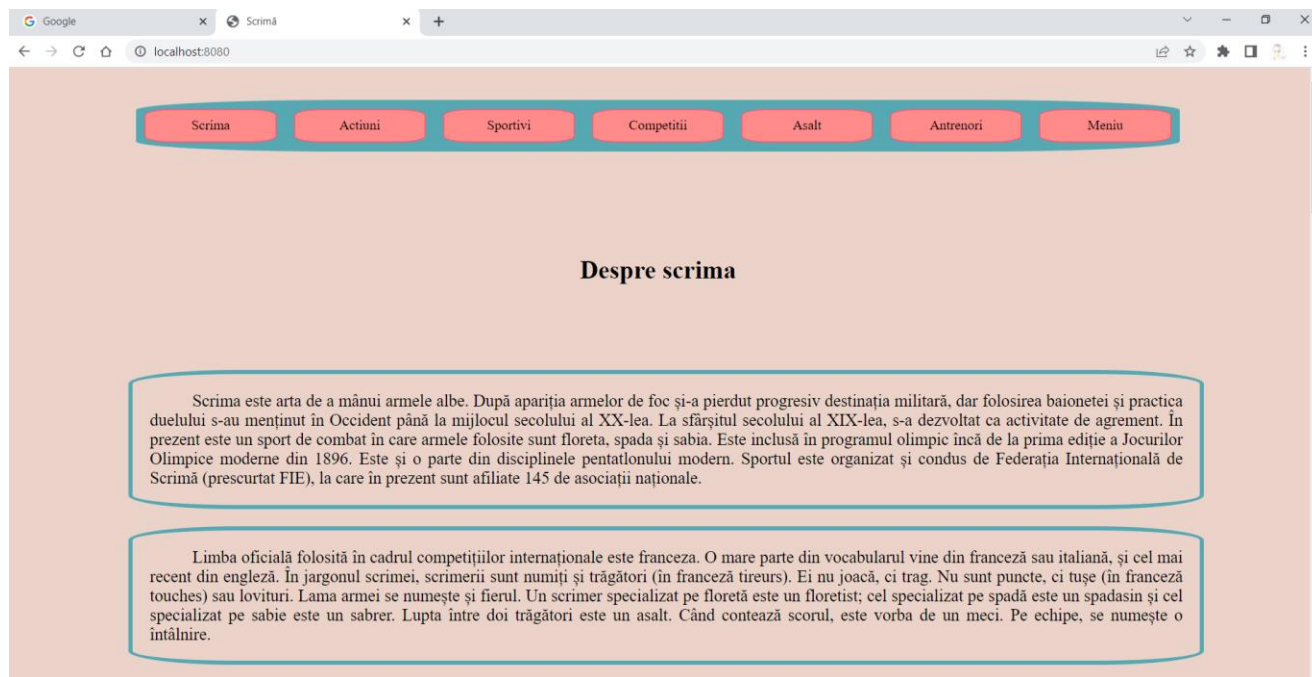
```

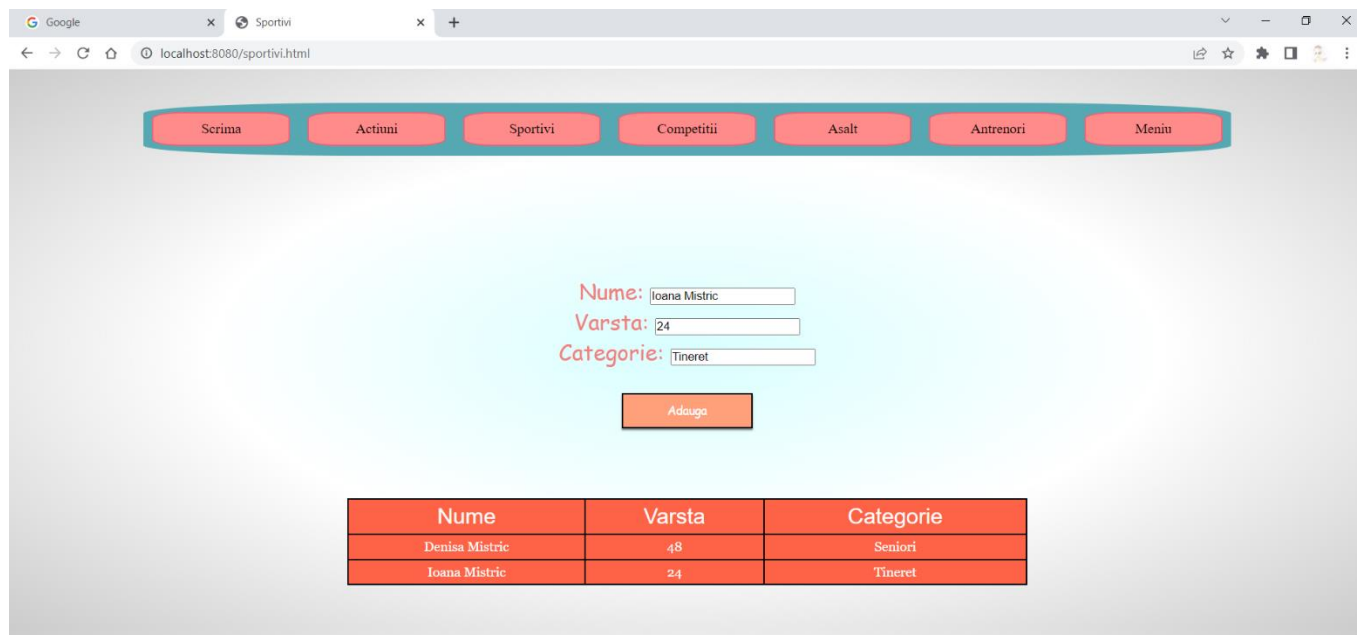
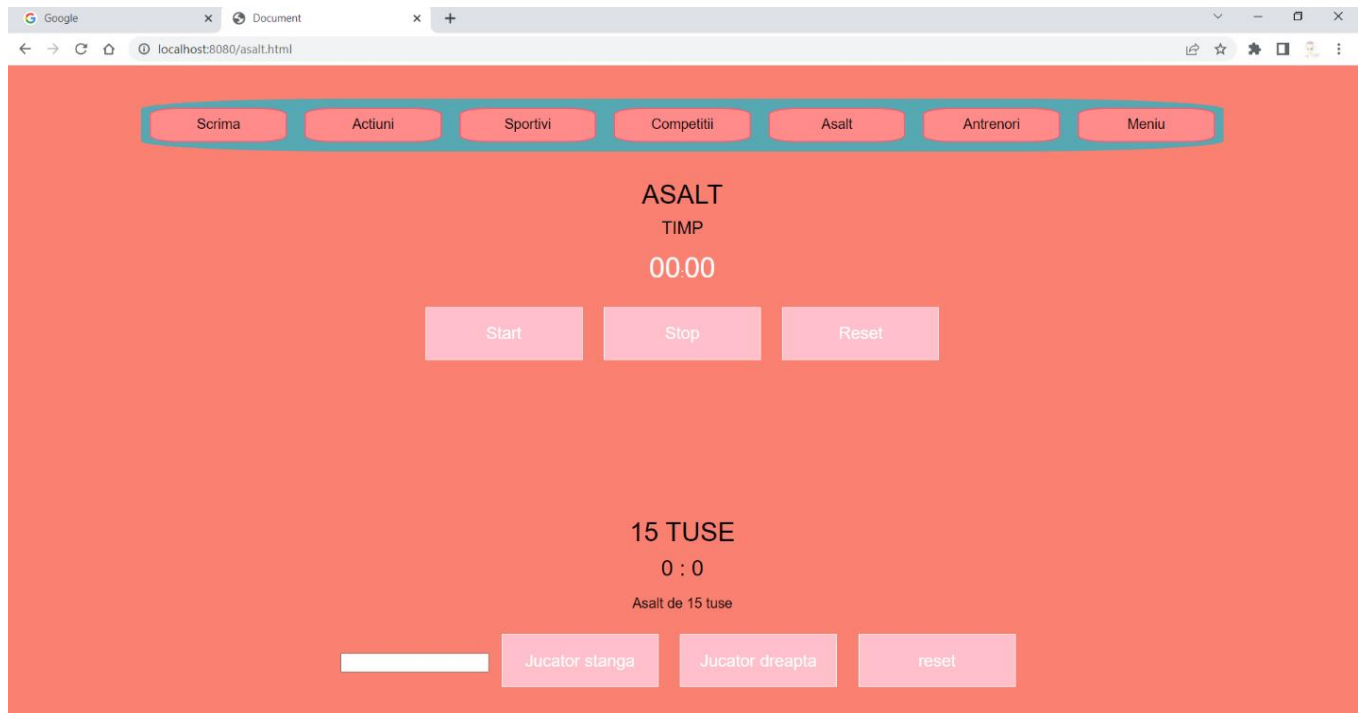
1  #include "Client.h"
2  #include "WebServer.h"
3
4  // using namespace std;
5
6  void main()
7  {
8      //Client client("0.0.0.0", 54000); //192.168.100.9
9      //if (client.init() != 0)
10     // return;
11
12     WebServer webserver("0.0.0.0", 8080);
13     //WebServer webserver("192.168.100.9", 8080);
14     if (webserver.init() != 0)
15         return;
16
17
18     webserver.run();
19
20     system("pause");
21 }

```

Ne putem conecta la localhost folosind "0.0.0.0" sau putem afla adresa IP a hostului, în cazul meu: "192.168.100.9".

- După ce vom porni serverul, clientul va putea să vadă paginile web disponibile:





Google

Scrimă

localhost:8080/index2.html

ScrimaActiuniSportiviCompetitiiAsaltAntrenoriMenu

Introduceti element:

Nume sportiv

Numele..

Prenume sportiv

Prenumele..

Nume adversar

Numele..

Prenume adversar

Prenumele..

Rezultat

Castigat

Submit

Capitolul 5. Server Linux

5.1. Noțiuni teoretice

Mecanismul de comunicare prin socket-uri conține câteva apeluri sistem. Apelul sistem socket stabilește punctul final al unei legături de comunicație.

sd = socket(format, type, protocol),

unde parametrul format precizează domeniul de comunicație (domeniul UNIX, sau domeniul Internet), type indică tipul de comunicație prin socket (circuit virtual, sau datagrama), iar protocol precizează un anumit protocol pentru controlul comunicației. În cadrul altor apeluri sistem, procesele vor întrebuința descriptorul de socket, sd. Apelul sistem close închide socket-urile.

Apelul sistem bind asociază un nume descriptorului de socket

bind(sd, address, length),

unde sd este descriptorul de socket, iar address este adresa unei structuri care precizează un indicator al domeniului și protocolului de comunicație, precizate în cadrul apelului sistem socket. Parametrul length reprezintă lungimea structurii de date address, fără acest parametru nucleul nu ar ști cât de lungă este adresa, deoarece lungimea acesteia poate diferi de la un domeniu (sau protocol) la altul. De exemplu, în cadrul domeniului UNIX, o adresă este un nume de fișier. Procesul server atașează adresele din apelul bind unor socket-uri și face publice numele lor pentru a fi identificate de către procesele client.

Apelul sistem connect cere nucleului să facă o conexiune cu un socket existent

connect(sd, address, length),

unde semnificația parametrilor este aceeași ca la apelul bind, cu deosebirea că parametrul address reprezintă adresa socket-ului destinație care va constitui celalalt capăt al liniei de comunicație. Ambele socket-uri trebuie să folosească același domeniu și protocol de comunicație, rămânând în sarcina nucleului inițializarea corectă a legăturilor de comunicație. Dacă tipul socket-ului este

datagrama, atunci apelul connect informează nucleul asupra adresei de utilizat în cadrul apelurilor send ulterioare prin socket-ul respectiv în momentul apelului nu se realizează nici o legătură.

Atunci când un proces server acceptă legăturile printr-un circuit virtual, nucleul trebuie să pună într-o coadă de așteptare cererile care sosesc, până în momentul în care va putea să le satisfacă. Apelul sistem listen precizează lungimea maximă a cozii de așteptare

```
listen(sd, qlength),
```

unde sd este descriptorul de socket și qlength reprezintă numărul maxim de cereri care vor fi luate în considerație.

Apelul sistem accept primește cererile de conectare la un proces server

```
nsd = accept(sd, address, addrlen),
```

unde sd este descriptorul de socket, address indică o zonă de date utilizator pe care nucleul o completează cu adresa de retur a procesului client care se conectează, iar addrlen precizează dimensiunea acestei zone. La revenirea din apelul accept, nucleul scrie în addrlen un număr care semnifică dimensiunea spațiului ocupat în zona de date. Apelul accept întoarce un nou descriptor de socket, nsd, diferit de descriptorul sd. Un proces server poate continua să asculte la socket-ul anunțat, în timp ce comunica cu un proces client pe un canal separat de comunicație.

Apelurile sistem send și recv permit transferul datelor printr-un socket. Sintaxa apelului sistem send este

```
count = send(sd, msg, length, flags),
```

unde sd este descriptorul de socket, msg este un pointer către datele care urmează să fie transmise, length reprezintă lungimea datelor de transmis, iar count este numărul de octeți efectiv transmiși.

5.2. Implementare

Procedurile pe care le-am implementat sunt:

- socket(): creare socket
- bind(): asociază un socket la un IP și un port unde poate
- listen(): listen pentru conexiuni
- accept(): accept conexiune
- send() sau receive(): trimite sau primește date la/de la socket-urile conectate

Implementare cod:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h> // pentru getnameinfo()

// Headere pentru socket-uri
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <arpa/inet.h>

#define SIZE 1024
#define BACKLOG 10 // Transmis catre listen()

void report(struct sockaddr_in* serverAddress);

void setHttpHeader(char httpHeader[])
{
    // Fisier obiect de returnat
    FILE* htmlData = fopen("actiuni.html", "r");

    char line[100];
    char responseData[8000];
    while (fgets(line, 100, htmlData) != 0) {
        strcat(responseData, line);
    }
    // char httpHeader[8000] = "HTTP/1.1 200 OK\r\n\n";
    strcat(httpHeader, responseData);
}
```

```

int main(void)
{
    char httpHeader[1000000] = "HTTP/1.1 200 OK\r\n\r\n";

    // Configurare socket: creeaza socket pentru comunicare, returneaza un
    descriptor
    // -----
    int serverSocket = socket(
        AF_INET,          // Domeniu: specifică familia de protocol
        SOCK_STREAM,      // Tip: specifică semantica comunicării
        0                 // Protocol: 0 deoarece exista un singur protocol pentru
familia specificata
    );

    // Structura locala de adrese
    // -----
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8001);
    serverAddress.sin_addr.s_addr = htonl(INADDR_LOOPBACK); //inet_addr("127.0.0.1");

    // Leg socket-ul la adresa locala
    // -----
    // bind() atribuie socket-ului adresa specificată de serverAddress
    // la care se face referire de către descriptorul de fișier serverSocket.
    bind(
        serverSocket,          // descriptor de fișier care se referă
la un socket
        (struct sockaddr*)&serverAddress, // Adresa care urmeaza sa fie atribuita
socket-ului
        sizeof(serverAddress) // Dimensiunea (octeți) a structurii
adresei
    );

    // Listen pentru conexiunile de intrare
    // -----
    int listening = listen(serverSocket, BACKLOG);
    if (listening < 0) {
        printf("Eroare: Serverul nu asculta.\n");
        return 1;
    }
    report(&serverAddress); // Funcție de raport personalizat
    setHttpHeader(httpHeader); // Funcție personalizată pentru a seta header-ul
    int clientSocket;

    // Aștept o conexiune, creez un socket conectat daca o conexiune este in
    asteptare
    // -----
    while (1) {
        clientSocket = accept(serverSocket, NULL, NULL);
        send(clientSocket, httpHeader, sizeof(httpHeader), 0);
        close(clientSocket);
    }
    return 0;
}

```

```

void report(struct sockaddr_in* serverAddress)
{
    char hostBuffer[INET6_ADDRSTRLEN];
    char serviceBuffer[NI_MAXSERV]; // definit in <netdb.h>
    socklen_t addr_len = sizeof(*serverAddress);
    int err = getnameinfo(
        (struct sockaddr*)serverAddress,
        addr_len,
        hostBuffer,
        sizeof(hostBuffer),
        serviceBuffer,
        sizeof(serviceBuffer),
        NI_NUMERICHOST
    );
    if (err != 0) {
        printf("Nu functioneaza!\n");
    }
    printf("\n\n\tServer-ul asculta: http://%s:%s\n", hostBuffer, serviceBuffer);
}

```

Observații:

Dacă `serverAddress.sin_addr.s_addr` este setat la `INADDR_ANY`, socket-ul este legat la toate interfețele locale.

`INADDR_ANY` este o constantă setată la zero, definită în `netinet/in.h`. Aceasta va corespunde unei adrese IP de 0.0.0.0 în notația standard IPv4. Se observa că `htonl(INADDR_LOOPBACK)` și `inet_addr("127.0.0.1")` sunt echivalente din punct de vedere funcțional.

Funcționalitate:

Pornesc server-ul:

```
ioana@ioana: ~/Desktop/ServerPSOproiect
ioana@ioana:~/Desktop/ServerPSOproiect$ ./server

Server-ul asculta: http://127.0.0.1:8001
```

Conectare la server (client) din browser:

