**Documentatie Server**

**1. Introducere**

Acest program implementează un server care procesează datele primite de la mai mulți clienți și calculează clasamente bazate pe scorurile competitorilor. Serverul realizează diverse sarcini, cum ar fi:

Primirea datelor competitorilor.

Procesarea și stocarea datelor.

Calcularea clasamentelor pentru țări și competitori.

Gestionarea transferurilor de fișiere.

Multi-threading pentru procesarea eficientă a datelor și gestionarea conexiunilor cu clienții.

Serverul este conceput să ruleze pe un sistem Windows utilizând Winsock pentru rețea și threading. Acesta suportă multiple conexiuni de clienți și procesează datele în mod concurent folosind un thread pool.

**2. Variabile Globale și Constante**

* **BUFFER\_SIZE**: Definește dimensiunea buffer-ului pentru primirea datelor de la clienți.
* **PORT**: Portul pe care serverul ascultă conexiunile clienților.
* **SERVER\_IP**: Adresa IP a serverului (localhost în acest caz).
* **serverRunning**: O variabilă atomică de tip boolean care indică dacă serverul trebuie să continue să ruleze.
* **allCompetitors**: Un vector global pentru a stoca toate datele competitorilor (id, scor, țară).
* **countryScores**: O mapă ce stochează scorul total pentru fiecare țară.
* **blacklist**: Un set care stochează ID-urile competitorilor care au fost blacklistati.
* **logMutex, countryScoresMutex, blacklistMutex**: Mutex-uri pentru a sincroniza accesul la resursele comune.
* **processingDone**: O variabilă care indică dacă procesarea datelor este completă.
* **cv**: O variabilă de condiție utilizată pentru sincronizare.
* **rankingsFuture**: O variabilă de tip future care este folosită pentru a calcula clasamentele în mod asincron.
* **lastRankingTime**: O variabilă folosită pentru a urmări ultima dată când au fost calculate clasamentele.

**3. Clasele de Bază**

**a. ThreadSafeQueue**

O clasă care implementează o coadă sigură pentru fire de execuție, pentru stocarea obiectelor de tip Competitor.

* **Metode:**
  + push(const Competitor& competitor): Adaugă un competitor în coadă.
  + pop(Competitor& competitor): Scoate un competitor din coadă, blocându-se dacă coada este goală.

**b. ThreadPool**

O clasă care implementează un thread pool pentru executarea concurentă a task-urilor.

* **Constructor:** Inițializează un thread pool cu un număr specificat de fire de execuție (numThreads).
* **Destructor:** Așteaptă și termină toate firele.
* **enqueue(function<void()> task):** Adaugă un task în coada pentru a fi executat de firele de execuție.

4. Funcții Ajutătoare

a. getCurrentTime()

Returnează timpul curent al sistemului sub formă de șir de caractere formatat. Este folosit pentru logare.

string getCurrentTime() {  
 auto now = chrono::system\_clock::now();  
 auto in\_time\_t = chrono::system\_clock::to\_time\_t(now);  
 stringstream ss;  
 ss << put\_time(localtime(&in\_time\_t), "%Y-%m-%d %X");  
 return ss.str();  
}

b. logMessage(const string& message)

Înregistrează mesaje într-un fișier (server\_log.txt) și în consolă. Folosește logMutex pentru a asigura accesul sincronizat la logare.

void logMessage(const string& message) {  
 lock\_guard<mutex> lock(logMutex);  
 ofstream logFile("server\_log.txt", ios::app);  
 logFile << "[" << getCurrentTime() << "] " << message << endl;  
 cout << "[" << getCurrentTime() << "] " << message << endl;  
}

c. addToBlacklist(const Competitor& competitor)

Adaugă un competitor în blacklist. ID-ul și țara competitorului sunt scrise în fișierul blacklist.txt.

void addToBlacklist(const Competitor& competitor) {  
 lock\_guard<mutex> lock(blacklistMutex);  
 blacklist.insert(competitor.id);  
 ofstream blacklistFile("blacklist.txt", ios::out);  
 blacklistFile << competitor.id << " " << competitor.country << "\n";  
 blacklistFile.close();  
 logMessage("[Server] Added competitor ID " + to\_string(competitor.id) + " to blacklist.");  
}

d. processCompetitorData(int deltaT)

Procesează datele competitorilor din coadă. Dacă scorul unui competitor este -1, acesta este adăugat în blacklist; altfel, scorul său este adăugat la scorul total al țării corespunzătoare.

void processCompetitorData(int deltaT) {  
 while (!processingDone) {  
 Competitor competitor;  
 if (competitorQueue.pop(competitor)) {  
 if (competitor.score == -1) {  
 addToBlacklist(competitor);  
 } else {  
 lock\_guard<mutex> lock(countryScoresMutex);  
 countryScores[competitor.country] += competitor.score;  
 logMessage("[Server] Processed competitor ID " + to\_string(competitor.id) + " from country " + competitor.country);  
 }  
 this\_thread::sleep\_for(chrono::milliseconds(deltaT));  
 }  
 }  
}

e. calculateRankings()

Calculează clasamentele pe baza scorurilor totale ale țărilor. Clasamentele sunt sortate în ordine descrescătoare a scorului.

vector<Ranking> calculateRankings() {  
 vector<Ranking> rankings;  
 {  
 lock\_guard<mutex> lock(countryScoresMutex);  
 for (const auto& entry : countryScores) {  
 rankings.push\_back({entry.first, entry.second});  
 }  
 }  
 sort(rankings.begin(), rankings.end(), [](const Ranking& a, const Ranking& b) {  
 return a.total\_score > b.total\_score;  
 });  
 logMessage("[Server] Calculated rankings.");  
 return rankings;  
}

f. saveRankingsToFile(const vector<Ranking>& rankings, const string& filename)

Salvează clasamentele (sortate după scorul țărilor) într-un fișier. Loghează clasamentul fiecărei țări și scorul său în fișierul specificat.

void saveRankingsToFile(const vector<Ranking>& rankings, const string& filename) {  
 // Specify the full path for the file  
 string fullPath = "D:\\An3Sem1\\Programare Paralela si Distributiva\\P1Tema\\" + filename;  
 ofstream outFile(filename);  
  
 // Print the rankings to the terminal for debugging  
 logMessage("[Server] Country Rankings:");  
 for (const auto& ranking : rankings) {  
 cout << ranking.country << ": " << ranking.total\_score << "\n";  
 // Write the rankings to the file  
 outFile << ranking.country << ": " << ranking.total\_score << "\n";  
 }  
  
 outFile.flush(); // Ensure the buffer is flushed  
 outFile.close(); // Ensure the file is properly closed  
 logMessage("[Server] Finished writing rankings to file: " + filename);  
}

g. saveCompetitorsToFile(vector<Competitor>& competitors, const string& filename)

Salvează lista de competitori cu ID-ul, scorul și țara acestora într-un fișier.

void saveCompetitorsToFile(vector<Competitor>& competitors, const string& filename) {  
 string fullPath = "D:\\An3Sem1\\Programare Paralela si Distributiva\\P1Tema\\" + filename;  
 ofstream outFile(filename);  
  
 logMessage("[Server] Competitors Ranking:");  
 for (const auto& competitor : competitors) {  
 cout << competitor.id << " " << competitor.score << " " << competitor.country << "\n";  
 outFile << competitor.id << " " << competitor.score << " " << competitor.country << "\n";  
 }  
  
 outFile.flush();  
 outFile.close();  
 logMessage("[Server] Finished writing competitors to file: " + filename);  
}

h. sendFile(SOCKET clientSocket, const string& filename)

Trimite un fișier specificat unui client conectat. Fișierul este citit în modul binar și trimis în bucăți către client.

void sendFile(SOCKET clientSocket, const string& filename) {  
 ifstream inFile(filename, ios::binary);  
 if (!inFile) {  
 logMessage("[Server] Error opening file: " + filename);  
 return;  
 }  
 char buffer[BUFFER\_SIZE];  
 while (inFile.read(buffer, BUFFER\_SIZE)) {  
 send(clientSocket, buffer, inFile.gcount(), 0);  
 }  
 inFile.close();  
 logMessage("[Server] Sent file: " + filename);  
}

i. handleClient(SOCKET clientSocket)

Gestionează comunicațiile cu un client. Primește datele competitorilor, le procesează și răspunde cu clasamentele finale atunci când este solicitat. De asemenea, ascultă mesaje speciale, cum ar fi "STOP\_SERVER" sau "GET\_FINAL\_RESULTS".

void handleClient(SOCKET clientSocket) {  
 char buffer[BUFFER\_SIZE];  
 int bytesReceived;  
 vector<Competitor> competitors;  
  
 // Receive data from client  
 while ((bytesReceived = recv(clientSocket, buffer, BUFFER\_SIZE - 1, 0)) > 0) {  
 buffer[bytesReceived] = '\0';  
 string data(buffer);  
 if (data == "DONE") {  
 logMessage("[Server] Received DONE message from client.");  
 break;  
 }  
 if (data == "STOP\_SERVER") {  
 logMessage("[Server] Received STOP\_SERVER message from client.");  
 serverRunning = false;  
 break;  
 }  
 istringstream stream(data);  
 string country;  
 int id, score;  
  
 while (stream >> id >> score >> country) {  
 // Check if competitor is blacklisted  
 {  
 lock\_guard<mutex> lock(blacklistMutex);  
 if (blacklist.find(id) != blacklist.end()) {  
 logMessage("[Server] Ignored blacklisted competitor ID " + to\_string(id) + " from country " + country);  
 continue;  
 }  
 }  
  
 // Push competitor data to queue  
 Competitor competitor{id, score, country};  
 competitorQueue.push(competitor);  
 allCompetitors.push\_back(competitor);  
 logMessage("[Server] Received data from " + country + ": " + to\_string(id) + " " + to\_string(score));  
 }  
 }  
  
 // Handle final results request  
 bytesReceived = recv(clientSocket, buffer, BUFFER\_SIZE - 1, 0);  
 buffer[bytesReceived] = '\0';  
 string request(buffer);  
 if (request == "GET\_FINAL\_RESULTS") {  
 logMessage("[Server] Received final results request from client.");  
  
 // Calculate and save final rankings  
 rankingsFuture = async(launch::*async*, calculateRankings);  
 vector<Ranking> rankings = rankingsFuture.get();  
 std::sort(allCompetitors.begin(), allCompetitors.end(), [](const Competitor& a, const Competitor& b) {  
 return a.score > b.score;  
 });  
 saveRankingsToFile(rankings, "final\_country\_rank.txt");  
 saveCompetitorsToFile(allCompetitors, "final\_competitors\_ranking.txt");  
  
 // Send final rankings files to client  
 sendFile(clientSocket, "final\_competitors\_ranking.txt");  
 sendFile(clientSocket, "final\_country\_rank.txt");  
 }  
  
  
 closesocket(clientSocket);  
}

j. compareSHA(const string& file1, const string& file2)

Compara două fișiere linie cu linie pentru a verifica dacă sunt identice. Este folosit pentru a valida corectitudinea rezultatelor generate de server prin compararea cu un fișier de rezultat așteptat.

//compare output function  
bool compareSHA(const string& file1, const string& file2) {  
 ifstream f1(file1);  
 ifstream f2(file2);  
 if (!f1 || !f2) {  
 cerr << "Error opening files." << endl;  
 return false;  
 }  
 string line1, line2;  
 while (true) {  
 bool f1End = !getline(f1, line1);  
 bool f2End = !getline(f2, line2);  
 if (f1End && f2End) {  
 return true;  
 }  
 if (f1End || f2End) {  
 return false;  
 }  
 if (line1 != line2) {  
 return false;  
 }  
 }  
}

**5. Funcția Principală**

Funcția principală realizează următoarele sarcini:

1. **Parsarea Argumentelor**: Serverul este configurat cu următoarele argumente din linia de comandă:
   * num\_readers: Numărul de fire pentru gestionarea clienților.
   * num\_workers: Numărul de fire pentru procesarea datelor competitorilor.
   * deltaT: Întârzierea în milisecunde între procesarea fiecărui competitor.
   * compare: Flag pentru a determina dacă trebuie să fie făcută comparația SHA la final.
2. **Inițializarea Winsock**: Inițializează biblioteca Winsock pentru rețea.
3. **Crearea și Asocierea Sockets**: Creează și asociază un socket serverului pe portul 8080.
4. **Crearea Thread Pool-ului**: Pornește thread pool-ul pentru procesarea datelor și gestionarea conexiunilor cu clienții.
5. **Acceptarea Conexiunilor Clienților**: Serverul acceptă conexiuni de la clienți și procesează datele fiecărei conexiuni într-un thread separat.
6. **Închidere Graceful**: Serverul așteaptă condiția de închidere (serverRunning = false), realizează curățarea și, opțional, compară fișierele de ieșire.
7. int main(int argc, char\* argv[]) {  
    if (argc < 5) {  
    cerr << "Usage: " << argv[0] << " <num\_readers> <num\_workers> <deltaT> <compare=1/0>" << endl;  
    return 1;  
    }  
     
    auto start = chrono::high\_resolution\_clock::now();  
     
    int PR = stoi(argv[1]);  
    int PW = stoi(argv[2]);  
    int DT = stoi(argv[3]);  
    int compare = stoi(argv[4]);  
     
    WSADATA wsaData;  
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {  
    cerr << "[Server] WSAStartup failed." << endl;  
    return 1;  
    }  
     
    SOCKET serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);  
    if (serverSocket == INVALID\_SOCKET) {  
    cerr << "[Server] Socket creation failed." << endl;  
    WSACleanup();  
    return 1;  
    }  
     
    sockaddr\_in serverAddr;  
    serverAddr.sin\_family = AF\_INET;  
    serverAddr.sin\_port = htons(PORT);  
    inet\_pton(AF\_INET, SERVER\_IP.c\_str(), &serverAddr.sin\_addr);  
     
    if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {  
    cerr << "[Server] Binding failed." << endl;  
    closesocket(serverSocket);  
    WSACleanup();  
    return 1;  
    }  
     
    if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR) {  
    cerr << "[Server] Listening failed." << endl;  
    closesocket(serverSocket);  
    WSACleanup();  
    return 1;  
    }  
     
    logMessage("[Server] Started on port 8080");  
     
    // Start the data processing threads  
    ThreadPool processingPool(PW);  
    for (int i = 0; i < PW; ++i) {  
    processingPool.enqueue([DT]() { processCompetitorData(DT); });  
    }  
     
    // Handle clients concurrently with thread pool  
    ThreadPool clientPool(PR);  
    for (int i = 0; i < PR; ++i) {  
    clientPool.enqueue([&]() {  
    while (serverRunning) {  
    SOCKET clientSocket = accept(serverSocket, NULL, NULL);  
    if (clientSocket == INVALID\_SOCKET) {  
    if (serverRunning) {  
    cerr << "[Server] Accepting connection failed." << endl;  
    }  
    continue;  
    }  
    logMessage("[Server] New client connected.");  
    handleClient(clientSocket);  
    }  
    });  
    }  
     
     
     
    // Wait for the server to stop  
    while (serverRunning) {  
    this\_thread::sleep\_for(chrono::milliseconds(100));  
    }  
     
     
     
    // Join all threads  
    processingDone = true;  
    cv.notify\_all();  
    closesocket(serverSocket);  
    WSACleanup();  
     
    if (compare == 1)  
    {  
    if (compareSHA("final\_competitors\_ranking.txt", "expected\_competitor.txt")) {  
    cout<<"[Server] Competitor Ranking == expected.\n";  
    }  
    else {  
    cout<<"[Server] Competitor Ranking != expected.\n";  
    }  
    if (compareSHA("final\_country\_rank.txt", "expected\_country.txt")) {  
    cout<<"[Server] Country Ranking == expected.\n";  
    }  
    else {  
    cout<<"[Server] Country Ranking != expected.\n";  
    }  
    }  
     
    if (!serverRunning)  
    {  
    exit(0);  
    }  
     
    return 0;  
   }

**6. Managementul Thread-urilor**

* **Thread-urile pentru procesarea datelor**: Sunt create PW fire de execuție pentru a procesa datele competitorilor în mod concurent. Fiecare fir procesează date din ThreadSafeQueue.
* **Thread-urile pentru gestionarea clienților**: Sunt create PR fire de execuție pentru a gestiona multiple conexiuni ale clienților în mod concurent. Fiecare fir ascultă pentru conexiuni și procesează datele clienților.

**7. Gestionarea Erorilor**

* Programul verifică erorile în timpul creării, asocierii și ascultării socket-ului.
* Erorile sunt înregistrate în server\_log.txt și afișate în consolă în cazul în care o operațiune critică eșuează.
* Dacă un competitor este pe blacklist, datele acestuia sunt ignorate și se face logare corespunzătoare.

**8. Gestionarea Fișierelor**

* Fișierele precum final\_country\_rank.txt, final\_competitors\_ranking.txt, și blacklist.txt sunt create/actualizate în timpul execuției pentru a stoca clasamentele și datele competitorilor.
* Fișierele sunt trimise către client pentru validare sau pentru rezultatele finale

**9. Utilizare**

Pentru a rula programul, se dau argumente din linia de comandă pentru configurare:

Server.exe <num\_readers> <num\_workers> <deltaT> <compare=1/0>

Unde:

* num\_readers: Numărul de fire pentru gestionarea clienților.
* num\_workers: Numărul de fire pentru procesarea datelor competitorilor.
* deltaT: Întârzierea (în milisecunde) între procesarea fiecărui competitor.
* compare: Setează 1 pentru a compara

**Documentație Client**

Acest program implementează un client care se conectează la un server pentru a trimite date ale competitorilor și pentru a primi rezultatele finale. Programul utilizează Winsock pentru a gestiona conexiunile de rețea și procesează datele într-un mod concurent, folosind thread-uri pentru trimiterea datelor și primirea fișierelor.

Serverul acceptă datele trimise de la client și calculează clasamente pe baza acestora. Clientul poate solicita, de asemenea, rezultatele finale și poate trimite comenzi pentru a opri serverul.

**1. Variabile Globale și Constante**

* **BUFFER\_SIZE**: Dimensiunea buffer-ului pentru trimiterea și primirea datelor.
* **PORT**: Portul pe care clientul se va conecta pentru a comunica cu serverul.
* **SERVER\_IP**: Adresa IP a serverului (în acest caz, 127.0.0.1 pentru localhost).
* **logMutex**: Mutex pentru a asigura accesul sincronizat la logarea mesajelor în fișierul de jurnal.

**2. Funcții Principale**

a. getCurrentTime()

Obține timpul curent al sistemului sub formă de șir de caractere formatat (e.g., 2025-01-09 12:30:00). Este folosit pentru a adăuga timestamp-uri în fișierele de logare.

string getCurrentTime() {  
 auto now = chrono::system\_clock::now();  
 auto in\_time\_t = chrono::system\_clock::to\_time\_t(now);  
 stringstream ss;  
 ss << put\_time(localtime(&in\_time\_t), "%Y-%m-%d %X");  
 return ss.str();  
}

b. logMessage(const string& message)

Loghează mesaje într-un fișier client\_log.txt, incluzând un timestamp pentru fiecare mesaj.

void logMessage(const string& message) {  
 lock\_guard<mutex> lock(logMutex);  
 ofstream logFile("client\_log.txt", ios::app);  
 logFile << "[" << getCurrentTime() << "] " << message << endl;  
}

c. sendData(SOCKET clientSocket, const unordered\_map<string, vector<pair<int, int>>>& data, int delay)

Trimite datele competitorilor către server în blocuri de câte 20 de competitori. Datele sunt trimise într-un interval de timp specificat de delay (în secunde). Fiecare bloc conține ID-ul competitorului, scorul acestuia și țara.

void sendData(SOCKET clientSocket, const unordered\_map<string, vector<pair<int, int>>>& data, int delay) {  
 for (const auto& countryData : data) {  
 const string& country = countryData.first;  
 const vector<pair<int, int>>& competitors = countryData.second;  
  
 for (size\_t i = 0; i < competitors.size(); i += 20) {  
 stringstream ss;  
 for (size\_t j = i; j < i + 20 && j < competitors.size(); ++j) {  
 ss << competitors[j].first << " " << competitors[j].second << " " << country << "\n";  
 }  
 string block = ss.str();  
  
 // Send data block to server  
 send(clientSocket, block.c\_str(), block.size(), 0);  
 logMessage("[Client] Sent data block to server: " + block);  
  
 this\_thread::sleep\_for(chrono::seconds(delay));  
 }  
 }  
  
 // Send DONE signal  
 string doneMessage = "DONE";  
 send(clientSocket, doneMessage.c\_str(), doneMessage.size(), 0);  
 logMessage("[Client] Sent DONE message to server.");  
}

d. receiveFile(SOCKET clientSocket, const string& filename)

Primește fișiere de la server și le salvează pe discul local. Fișierele sunt primite în mod binar și scrise în fișierele locale.

void receiveFile(SOCKET clientSocket, const string& filename) {  
 ofstream outFile(filename, ios::binary);  
 char buffer[BUFFER\_SIZE];  
 int bytesReceived;  
 while ((bytesReceived = recv(clientSocket, buffer, BUFFER\_SIZE, 0)) > 0) {  
 outFile.write(buffer, bytesReceived);  
 }  
 outFile.close();  
 logMessage("[Client] Received file: " + filename);  
}

e. requestFinalResults(SOCKET clientSocket)

Trimite un mesaj GET\_FINAL\_RESULTS serverului pentru a solicita rezultatele finale. După trimiterea cererii, clientul primește fișierele rezultate (clasamentele finale ale competitorilor și țărilor).

void requestFinalResults(SOCKET clientSocket) {  
 string request = "GET\_FINAL\_RESULTS";  
 send(clientSocket, request.c\_str(), request.size(), 0);  
 logMessage("[Client] Requested final results from server.");  
  
 // Receive final results files  
 receiveFile(clientSocket, "final\_competitors\_ranking.txt");  
 receiveFile(clientSocket, "final\_country\_ranking.txt");  
}

f. loadData(const string& filename)

Încarcă datele dintr-un fișier specificat (de exemplu, Italy.txt), citind ID-ul competitorului, scorul acestuia și țara. Aceste date sunt stocate într-un unordered\_map în care cheia este țara și valoarea este o listă de perechi (ID, scor).

unordered\_map<string, vector<pair<int, int>>> loadData(const string& filename) {  
 unordered\_map<string, vector<pair<int, int>>> data;  
 ifstream inputFile(filename);  
 if (!inputFile) {  
 cerr << "Error opening file: " << filename << endl;  
 return data;  
 }  
  
 int id, score;  
 string country;  
 while (inputFile >> id >> score >> country) {  
 data[country].emplace\_back(id, score);  
 }  
 return data;  
}

g. sendStopServerMessage()

Trimite un mesaj STOP\_SERVER serverului pentru a opri execuția acestuia. Este folosit doar în cazul în care clientul este asociat cu un fișier special (de exemplu, Italy.txt).

void sendStopServerMessage() {  
 WSADATA wsaData;  
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {  
 cerr << "WSAStartup failed." << endl;  
 return;  
 }  
  
 SOCKET clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (clientSocket == INVALID\_SOCKET) {  
 cerr << "Socket creation failed." << endl;  
 WSACleanup();  
 return;  
 }  
  
 sockaddr\_in serverAddr;  
 serverAddr.sin\_family = AF\_INET;  
 serverAddr.sin\_port = htons(PORT);  
 inet\_pton(AF\_INET, SERVER\_IP.c\_str(), &serverAddr.sin\_addr);  
  
 if (connect(clientSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {  
 cerr << "Connection to server failed." << endl;  
 closesocket(clientSocket);  
 WSACleanup();  
 return;  
 }  
  
 const char\* stopMessage = "STOP\_SERVER";  
 send(clientSocket, stopMessage, strlen(stopMessage), 0);  
  
 closesocket(clientSocket);  
 WSACleanup();  
}

**4. Funcția Principală (main)**

Funcția principală realizează următoarele activități:

1. **Inițializarea Winsock**: Încarcă biblioteca Winsock pentru a iniția conexiuni de rețea.
2. **Crearea Socket-ului Clientului**: Crează un socket de tip TCP pentru comunicarea cu serverul.
3. **Conectarea la Server**: Încercă să se conecteze la server pe portul specificat.
4. **Înregistrarea în Loguri**: La fiecare pas semnificativ, clientul scrie mesaje în fișierul client\_log.txt pentru a urmări starea și activitatea clientului.
5. **Încărcarea Datelor**: Datele sunt încărcate din fișierul specificat ca argument al liniei de comandă.
6. **Trimiterea Datelor**: Datele sunt trimise către server prin funcția sendData.
7. **Cererea Rezultatelor Finale**: Clientul cere rezultatele finale ale competiției prin funcția requestFinalResults.
8. **Închiderea Conexiunii**: După primirea rezultatelor, conexiunea la server este închisă.
9. **Oprirea Serverului (opțional)**: Dacă fișierul încărcat este Italy.txt, clientul va trimite un mesaj pentru a opri serverul.
10. int main(int argc, char\* argv[]) {  
     if (argc < 2) {  
     cerr << "Usage: " << argv[0] << " <file\_path>" << endl;  
     return 1;  
     }  
      
     string filepath = argv[1];  
      
      
     WSADATA wsaData;  
     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {  
     cerr << "[Client] WSAStartup failed." << endl;  
     return 1;  
     }  
      
     SOCKET clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);  
     if (clientSocket == INVALID\_SOCKET) {  
     cerr << "[Client] Socket creation failed." << endl;  
     WSACleanup();  
     return 1;  
     }  
      
     sockaddr\_in serverAddr;  
     serverAddr.sin\_family = AF\_INET;  
     serverAddr.sin\_port = htons(PORT);  
     inet\_pton(AF\_INET, SERVER\_IP.c\_str(), &serverAddr.sin\_addr);  
      
     if (connect(clientSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {  
     cerr << "[Client] Connection failed." << endl;  
     closesocket(clientSocket);  
     WSACleanup();  
     return 1;  
     }  
      
     logMessage("[Client] Connected to server.");  
      
     // Load data from file  
     unordered\_map<string, vector<pair<int, int>>> data = loadData(filepath);  
      
     // Send data to server  
     sendData(clientSocket, data, 1);  
      
     // Request and receive final results  
     requestFinalResults(clientSocket);  
      
     closesocket(clientSocket);  
     WSACleanup();  
      
     // Send STOP\_SERVER message  
     if (filepath == "Italy.txt")  
     sendStopServerMessage();  
      
     return 0;  
    }

**5. Utilizare**

Pentru a rula clientul, trebuie să specifici fișierul de intrare care conține datele competitorilor:

Client.exe <file\_path>

Exemplu de utilizare:

Client.exe Italy.txt

Unde file\_path este calea către fișierul care conține datele de intrare (de exemplu, Italy.txt).

**6. Gestionarea Erorilor**

* **Inițializarea Winsock**: Dacă inițializarea Winsock eșuează, clientul va afișa un mesaj de eroare și va opri execuția.
* **Crearea Socket-ului**: Dacă crearea socket-ului eșuează, clientul va înregistra o eroare și va închide aplicația.
* **Conectarea la Server**: Dacă conexiunea la server nu este posibilă, clientul va înregistra o eroare și va opri execuția.
* **Încărcarea Fișierului**: Dacă fișierul de date nu poate fi deschis, clientul va afișa un mesaj de eroare.

**7. Observații și Extensii**

* **Rata de trimitere a datelor**: Clientul trimite datele competitorilor în blocuri de câte 20 de competitori. Aceasta poate fi ajustată pentru a trimite mai multe sau mai puține date simultan.
* **Cererea de rezultate**: Clientul primește fișierele de clasamente în format text, care sunt stocate local. Fișierele primite sunt denumite final\_competitors\_ranking.txt și final\_country\_ranking.txt.
* **Comanda de oprire a serverului**: Este trimisă automat la finalizarea procesului, în cazul în care fișierul încărcat este Italy.txt.

Această arhitectură permite unui client să interacționeze cu serverul pentru a trimite date și a solicita rezultatele finale, menținând un flux controlat de date și operând în mod sincronizat.

**Cazuri de Testare**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nr. Reader Threads** | **Nr. Workers** | **Timp executie mediu** | **DeltaT** |
| 4 | 4 | 10474.1 | 1ms |
| 4 | 4 | 8905.85 | 2ms |
| 4 | 4 | 13359.81 | 4ms |
| 2 | 2 | 15349.51 | 1ms |
| 2 | 2 | 15337.9 | 2ms |
| 2 | 2 | 17262.98 | 4ms |
| 4 | 2 | 17255.49 | 1ms |
| 4 | 2 | 19170 | 2ms |
| 4 | 2 | 19236.14 | 4ms |
| 4 | 8 | 17233.98 | 1ms |
| 4 | 8 | 13460.84 | 2ms |
| 4 | 8 | 15322.48 | 4ms |