LocalMarketplacePlatform

Cristian Cristescu

December 2023

1 Introducere

Proiectul vizeaza crearea unei platforme locale de tip marketplace. Aceasta va permite utilizatorilor sa isi creeze conturi pentru a cumpara sau vinde produse in comunitatea lor, folosind un server central pentru gestionarea tranzactiilor si a listelor de produse. Utilizatorii vor avea la dispozitie un minim de 10 operatii pentru manipularea articolelor si a propriilor conturi.

2 Tehnologii Utilizate

2.1 TCP - Transmission Control Protocol

Protocolul **TCP** a fost folosit in realizarea acestui proiect pentru a asigura integritate in transmiterea datelor de la client la server si invers.

TCP este un protocol de transport orientat pe conexiune ce ofera incredere, asigura livrarea ordonata a unui flux de octeti de la un endpoint din retea la un alt endpoint din retea. Pe langa sarcinile sale de gestionare a traficului, TCP controleaza marimea segementului de date, debitul de informatie, rata la care se face schimbul de date, precum si evitarea congestionarii traficului de retea. TCP poate fi vazut si ca fiind un serviciu de incredere care garanteaza livrarea unui flux de date transmis de la o gazda la alta fara duplicarea sau pierderea de date

2.2 SQLite3 C/C++ API

Pentru a stoca articolele si datele utilizatorilor de pe platforma am folosit o baza de date sqlite si API-ul SQLite3 pentru c/c++ pentru conectarea la baza de date si efectuarea diferitelor operatii supra datelor. Pentru a putea folosi acest API am folosit libraria ”sqlite3.h” ce facilitează crearea, manipularea și gestionarea datelor stocate în baza de date SQLite într-un mod simplu și flexibil. API-ul oferă funcții și proceduri pentru a inițializa și stabili o conexiune cu baza de date SQLite. Utilizatorii pot accesa cu ușurință structurile și funcționalitățile necesare pentru a începe lucrul cu datele.

3 Arhitectura Aplicatiei

3.1 Arhitectura Clientului

- Clientul apeleaza primitiva socket pentru a se conecta la server, salveaza descriptorul creat si se coneacteaza la server prin intermediul primitivei connect.
- Apoi citeste din terminal comenzile, le transmite catre server si afiseaza mesajul aferent comenzii furnizat de server.

3.2 Arhitectura Serverului

- Serverul apeleaza primitiva socket, salveaza descriptorul creat, populeaza structura de date folosita de server, si ataseaza socket-ului informatiile respective folosind primitva bind.

- Asculta prin intermediul primitivei listen daca se conecteaza vreun client, intra intr-o bucla infinita in cadrul careia accepta conexiunile clientilor prin intermediul primitive accept.
- Retine descriptorul creat de apelul accept care va fi folosit ulterior pentru comunicarea clien-server si populeaza structura de date ce va fi transmisa thread-ului.
- Pentru fiecare client conectat, server-ul creeaza cate un thread folosind funtia pthreadcreate care se va ocupa de parsarea si executarea comenzilor.
- In handler-ul thread-ului, treat(), intra intr-o bucla infinita in cadrul careia parseaza comanda, realizeaza conexiunea cu baza de date, extrage datele necesare si genereaza raspunsul aferent input-ului.
- Structura bazei de date consta in 3 tabele, unul in care sunt retinute date despre utilizatori pentru partea de autentificare, un tabel contine datele despre articole(titlu, descriere, pret, status, categorie si id-ul proprietarului), iar al treilea table reprezinta o asociere dintre tabela de utilizatori si tabela cu articole pentru a ilustra si stoca istoricul de achizitii al fiecarui utilizator.
- Serverul va lucra cu 4 clase: Client, Article, Database, Command.

3.3 Diagrama Aplicatiei

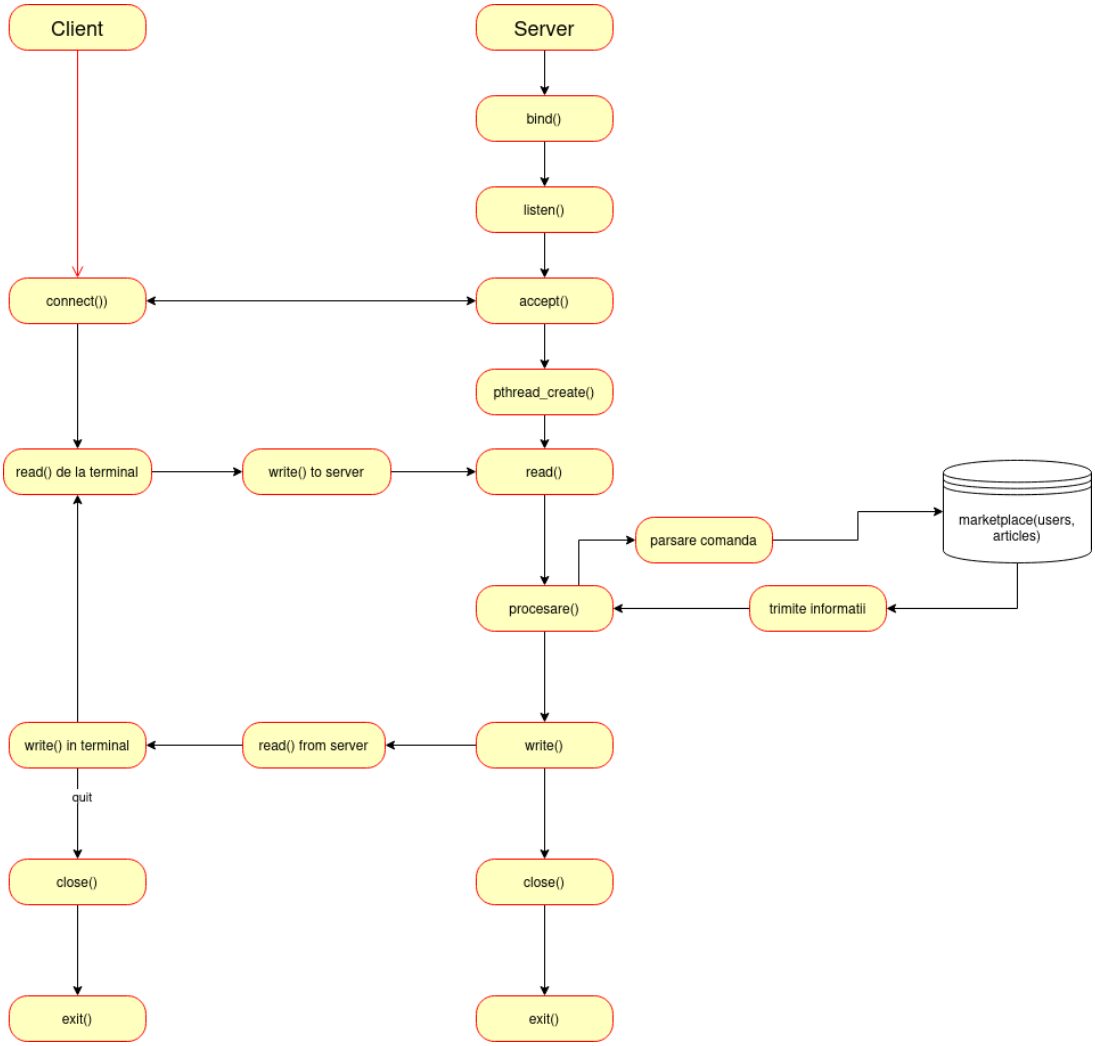

Figure 1: Structura Programului

4 Detalii de implementare

Intreaga comunicare dintre clienti si server consta in urmatorul set de comenzi:

- help: va afisa comenzile suportate pe acest server
- login "username" "password": intra in contul corespunzator username-ului si parolei
- register "username" "password": creeaza un cont pe acest server daca nu exista un alt user cu acelasi username
- add_article "title" "description" "price" "category": creeaza un nou articol
- get_all_articles "flag" "arguments":
  - "-a": printeaza toate articolele disponibile
  - "-u": printeaza toate articolele aflate in posesia user-ului logat
  - "-c" "category": filtreaza si printeaza articolele care au aceeasi categorie ca si cea furnizata
  - "-p" "down" "up": filtreaza si printeaza articolele care au un pret cuprins intre down si up.
- update_article "flags" "articleId" "argument":
  - "-t": inlocuieste titlul articolului cu id-ul specificat cu argumentul furnizat
  - "-d": inlocuieste descrierea articolului cu id-ul specificat cu argumentul furnizat
  - "-p": inlocuieste pretul articolului cu id-ul specificat cu argumentul furnizat
  - "-c": inlocuieste categoria articolului cu id-ul specificat cu argumentul furnizat
- get_purchase_history: printeaza istoricul achizitiilor unui user daca acesta este logat
- remove "id": sterge un articol identificat prin id-ul furnizat, daca acesta apartine user-ului logat pe server
- logout: deconecteaza user-ul de la server
- quit: paraseste serverul

4.1 Client

In cadrul implementarii clientului am folosit primita socket() pentru comunicarea server client.

```
if ((sd = socket( domain: AF_INET,  type: SOCK_STREAM,  protocol: 0)) == -1) {
    perror( s: "Socket error.\n");
    return errno;
}
```

Figure 2: Client Socket

Apoi urmeaza populatarea structurii de date server cu IP-ul si portul furnizat la lansarea aplicatiei client si conectarea la server prin intermediul apelului connect():

```
server.sin_family = AF_INET;
server.sin_addr.s_addr = inet_addr( cp: argv[1]);
server.sin_port = htons( hostshort: port);

if (connect( fd: sd,  addr: (struct sockaddr *) &server,  len: sizeof(struct sockaddr)) == -1) {
    perror( s: "[client]Error at connect().\n");
    return errno;
}
```

Figure 3: Connecting to server

Daca nu au existat erori pana in acest punct, clientul va intra intr-o bucla infinita in care va citi de la terminal comenzile, le va trimite la server si va astepta pentru un raspuns, apoi il afiseaza. Aplicatia client se va inchide la executarea comenzii "quit" sau daca server-ul a fost inchis.

4.2 Server

In cazul serverului, apelul primitivei socket si popularea structurii server, se realizeaza intr-o maniera asemanatoare. In schimb, sunt apelate inca 2 primitive, bind() si listen(), inainte de conectarea efectiva cu clientii.

```
if (bind( fd: sd,  addr: (struct sockaddr *) &server,  len: sizeof(struct sockaddr)) == -1) {
    perror( s: "[server]Bind error.\n");
    return errno;
}

if (listen( fd: sd,  n: 5) == -1) {
    perror( s: "[server]Listen error.\n");
    return errno;
}
```

Figure 4: Preparing the server

Acceptarea efectiva a clientilor este realizata intr-o bulca infinita, prin intermediul primitivei accept. Dupa ce este realizata conexiunea, serverul creeaza cate un thread corespunzator fiecarui client conectat ce va servi cererile clientului respenctiv.

```
int client;
thData *td;
int length = sizeof(from);

printf( format: "[server]Waiting on the port: %d...\n", PORT);
fflush( stream: stdout);

if ((client = accept( fd: sd,  addr: (struct sockaddr *) &from,  addr_len: reinterpret_cast<socklen_t *>(&length))) < 0) {
    perror( s: "[server]Accept error.\n");
    continue;
}

td = new thData( id: i,  c: client,  d: db,  us: new Client( clientSocket: client),  &: mutex);

pthread_create( newthread: &th[i],  attr: nullptr,  start_routine: &treat,  arg: td);
i++;
```

Figure 5: Accepting clients connections

```
struct thData {
    int idThread;
    int cl;
    Database *db;
    Client *user;
    std::mutex &mutex;

    thData(int id, int c, Database *d,
           Client *us, std::mutex &m) : mutex(m),
           idThread(id), cl(c), db(d), user(us) {};
};
```

Figure 6: The data structure passed to every thread

In cadrul functiei treat, apelez functia execute ce va realiza comunicarea efectiva cu clientul, receptionarea comenzilor, parsarea lor si transmiterea mesajului aferent.

Parsarea comezilor o realizez cu ajutorul expresiilor regulate, deoarece comenzile pot contine spatii in argumentele lor. Atunci cand serverul primeste o comanda, creez un obiect de tipul Command ce va primi ca argument comanda efectiva si va introduce fiecare token intr-un vector.

```
Command::Command(char* command) {
    this->command = command;
    std::regex rgx( p: "\"([^\"]*)\"|(\\S+)");
    std::smatch match;
    std::string com = command;
    int i = 0;

    while (std::regex_search( s: com,  &: match,  e: rgx)) {
        std::string token;
        if(i == 0){
            token = match[0].str();
        }else{
            token = match[1].str();
        }
        if (!token.empty()) {
            this->tokens.push_back(token);
        }
        com = match.suffix().str();
        i++;
    }
}
```

Figure 7: Parsing the command

Comunicarea cu baza de date o realizez printr-un obiect global partajat de tipul Database, ce primeste un path catre fisierul ce contine baza de date si genereaza conexiuni catre aceasta.

Fiecare operatie ce presupune operatii cu baza de date este protejata de un lock: std::mutex ce este partajat cu fiecare thread. Folosirea unui lock ajuta la preventia eventualelor erori de genul: daca un client doreste sa afiseze toatea articolele disponibile si un altul insereaza un articol nou, pot aparea conflicte la accesarea bazei de date si operatiile nu vor mai putea fi efectuate.

5 Concluzii

Proiectul propune sa ofere utilizatorilor un mod simplu si usor de a vinde si cumpara diferite articole intr-o comunitate locala.

Pentru a imbunatati appicatia, pe viitor se poate implementa o interfata grafica pentru client, pentru a oferi utilizatorilor un mod mai eficient si simplu de a interactiona cu aplicatia.

Bibliografie

- https://profs.info.uaic.ro/ computernetworks/index.php
- https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- https://app.diagrams.net/
- https://www.sqlite.org/docs.html