# Service Registry (B) - Documentație proiect

Ciorâțanu Maria, grupa A5, anul III

Facultatea de Informatică Iași

Ianuarie 2025

**Abstract.** Acest document reprezintă documentația (raportul tehnic) proiectului **Service Registry**, care funcționează ca un registru centralizat pentru descoperirea serviciilor într-o rețea de calculatoare. Soluția oferă un mod de a înregistra, actualiza, căuta și șterge servicii, facilitând totodată mecanisme de *autentificare* și *monitorizare* prin log-uri. Astfel, aplicația dezvoltă un fel de **service discovery** (similar cu orchestratoare precum Kubernetes), folosind un protocol de comunicare **TCP** și mesaje **JSON**.
Proiectul a fost dezvoltat folosind limbajele **C/C++**, sub un sistem de operare tip **Linux/UNIX**, și include atât un *server concurent*, cât și un *client* de test. Pe lângă îndeplinirea cerințelor de bază (înregistrare, reînnoire, listare, autentificare), aplicația permite și monitorizarea și **logarea** tuturor operațiilor.

## 1 Introducere

**ServiceRegistry (B)**

Creati o aplicatie server care funcționează ca un registru de servicii distribuite, pentru descoperirea serviciilor într-o rețea de calculatoare. Serverul va gestiona o listă de servicii active, fiecare identificat printr-un nume unic, o adresă IP și un port. Fiecare serviciu înregistrat trebuie să aibă un timp de expirare (TEx) specificat, după care este eliminat automat din registru dacă nu este reînnoit. Serverul va expune următoarele funcționalități printr-un API simplu: inregistrare serviciu (clienții pot înregistra un nou serviciu specificând parametrii necesari (nume, IP, port, TEx)), reînnoire serviciu (fiecare serviciu își reînnoiește înregistrarea la intervale regulate pentru a rămâne activ), cerere de adresă (clienții pot obține adresa IP și portul unui serviciu activ folosind numele serviciului), listare servicii active (clienții pot obține lista completă a serviciilor active, cu toate informațiile (nume, IP, port, timp rămas până la expirare)). Serverul trebuie să fie capabil să simuleze un comportament similar unui orchestrator de containere (e.g. Kubernetes sau Podman), oferind funcționalități de "service discovery". În acest context, orice client poate întreba serverul pentru adresa oricărui serviciu activ și poate obține adresa serviciului cel mai recent actualizat. Serverul va funcționa în mod concurent, comunicând cu clienții prin TCP și folosind mesaje JSON. Pentru testare, dezvoltați un client simplu care poate înregistra, actualiza, șterge și solicita informații despre servicii, simulând funcționarea mai multor servicii care își reînnoiesc automat TEx-ul. De asemenea, se va implementa un mecanism de autentificare, prin care doar clienții autorizați pot înregistra sau actualiza servicii, si functionalitati de logare și monitorizare, astfel încât fiecare cerere primită de server să fie înregistrată într-un fișier de log (cu timestamp, IP, port, acțiune).

Proiectul *Service Registry* reprezintă o aplicație client-server care gestionează **servicii** într-o rețea de calculatoare. Din perspectiva disciplinei *Rețele de Calculatoare*, am aplicat elemente precum:

- comunicare **TCP/IP** între client și server

- concurență la nivel de server folosind **thread**-uri POSIX (`pthread`)
- schimb de mesaje în format **JSON** pentru transferul cererilor și răspunsurilor
- **logare** acțiuni (crearea unui fișier de tip `server.log` care urmărește activitatea)
- **SQLite** pentru stocarea și actualizarea serviciilor și a conturilor de utilizator

Aplicația se comportă ca un registru centralizat: fiecare serviciu înregistrat trebuie să specifice un `TEx` (timp de expirare), după care este *inactivat* automat dacă nu se reînnoiește. Astfel, **serverul** are și rolul de a monitoriza aceste servicii, lansând thread-uri care simulează activitatea fiecărui serviciu și verificând dacă acestea au expirat.

### 1.1 Obiective

- **Stocare organizată** – folosind SQLite pentru a gestiona cu ușurință informațiile despre servicii și utilizatori.
- **Concurență** – server multi-threaded care gestionează simultan multiple conexiuni.
- **Interfață client intuitivă** – meniu simplu pentru înregistrare, autentificare, administrarea serviciilor.
- **Logare și monitorizare** – fișier `server.log` și fișiere `service_<name .txt` pentru fiecare serviciu.

## 2 Tehnologii Aplicate

### 2.1 Protocolul TCP

Am ales **TCP** datorită fiabilității sale. Mesajele JSON trebuie să ajungă integral și în ordinea corectă, fapt ce ne asigură o comunicare sigură, importantă în orice aplicație de rețea care gestionează date critice (autentificări, actualizări de servicii etc.). De asemenea, pe server folosesc funcții de sistem POSIX (precum `socket()`, `bind()`, `listen()`, `accept()`) pentru inițierea și menținerea conexiunilor cu clienții.

### 2.2 Mesaje JSON & Protocol

Pentru a structura clar datele transmise, aplicația folosește un **protocol JSON custom** între client și server.

#### Structura Generală a Cererilor

```
{
    "object": "<object_type>",
    "command": "<command_type>",
    "attributes": { "key": "value", ... }
}
```

**Structura Generală a Răspunsurilor**

```
{
    "status": "<ok | error>",
    "message": "<message_content>"
}
```

- **status**: poate fi `ok` (cerere reușită) sau `error` (apare o eroare).
- **message**: conține datele efective (un string, un obiect JSON sau un array JSON).

**Exemple de Comenzi Specifice**:

- **1. Register User**

```
{
  "object": "user",
  "command": "register",
  "username": "test_user",
  "password": "test_password",
  "role": "user"
}
```

- **2. Login User**

```
{
  "object": "user",
  "command": "login",
  "username": "test_user",
  "password": "test_password"
}
```

- **3. Register Service**

```
{
  "object": "service",
  "command": "register",
  "name": "example_service",
  "ip": "127.0.0.1",
  "port": 8080,
  "tex": 3600
}
```

- **4. Delete Service**

```
{
  "object": "service",
  "command": "delete",
  "name": "example_service"
}
```

**Exemplu de Răspuns de Succes**:

```
{
  "status": "ok",
  "message": "Service added successfully"
}
```

**Exemplu de Răspuns cu Eroare**:

```
{
  "status": "error",
  "message": "User already exists"
}
```

# 3 Structura Aplicației

Proiectul este organizat după cum urmează:

## 3.1 Componente Principale

- **Client** (`client.cpp`)
  Inițiază conexiunea la server (TCP), gestionează *autentificarea* (login/register) și oferă un meniu pentru:
  - **register_service()** - Înregistrarea unui nou serviciu.
  - **update_service()** - Actualizarea/Reînnoirea unui serviciu.
  - **search_service()** - Obținerea detaliilor unui serviciu după nume.
  - **delete_service()** - Ștergerea unui serviciu din registru.
  - **get_last_updated_service()** - Afișarea serviciului cu ultimul update efectuat.
  - **get_all_services()** - Listarea tuturor serviciilor disponibile.
  - **quit()** - Deconectarea de la server.

```cpp
while (true)
{
    std::cout << "Please choose an option:\n";
    std::cout << "1. Register a service\n";
    std::cout << "2. Search for a service\n";
    std::cout << "3. Update a service\n";
    std::cout << "4. Delete a service\n";
    std::cout << "5. Get last updated service\n";
    std::cout << "6. Get all services\n";
    std::cout << "7. Quit\n";

    std::string choice;
    std::getline(std::cin, choice);

    if (choice == "1")
    {
        register_service();
    }
    else if (choice == "2")
    {
        search_service();
    }
    else if (choice == "3")
    {
        update_service();
    }
    else if (choice == "4")
    {
        delete_service();
    }
    else if (choice == "5")
    {
        get_last_updated_service();
    }
    else if (choice == "6")
    {
        get_all_services();
    }
    else if (choice == "7")
    {
        quit();
    }
```

**Fig. 1.** Fragment de cod din `client.cpp`.

- **Server** (`server.cpp`)
  - Ascultă pe portul **2025**.
  - Creează un thread pentru fiecare conexiune de client, folosind `pthread_create`.
  - Folosește **Controller**, **UserHandler**, **ServiceHandler** pentru logica de business.

- Folosește **UserRepository** și **ServiceRepository** (Singletoane) pentru operațiile CRUD în SQLite.
- Loghează toate cererile în `server.log`, inclusiv date despre client (IP, port, thread_id).

```
while (1)
{
    int client;
    thData *td;
    socklen_t length = sizeof(from);

    // accept incoming client connection
    if ((client = accept(sd, (struct sockaddr *)&from, &length)) < 0)
    {
        std::cerr << "[server] Error at accept()\n";
        continue;
    }

    // convert client IP to string
    char client_ip_str[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &from.sin_addr, client_ip_str, INET_ADDRSTRLEN);
    int client_port = ntohs(from.sin_port);

    std::cout << "[server] Accepted client: " << client_ip_str << ":" << client_port << "\n";

    // allocate thData using new (replace malloc)
    td = new thData();
    td->idThread = i++;
    td->cl = client;
    td->client_ip = std::string(client_ip_str);
    td->client_port = client_port;
    td->isLoggedIn = false;

    // create a detached thread to handle the client
    if (pthread_create(&th, NULL, &LinuxAPIServerCommunication::treat, td) != 0)
    {
        std::cerr << "[server] Error creating thread\n";
        close(client);
        delete td; // replace free with delete
    }
    // no need to store pthread_t since threads are detached
}
```

**Fig. 2.** Fragment din `server.cpp`, bucla principală de acceptare conexiuni.

- **JsonParser** (`JsonParser.h & JsonParser.cpp`)
  Implementare minimală pentru parsarea obiectelor JSON (tipuri: `STRING`, `NUMBER`, `BOOL`, `OBJECT`, `ARRAY`). Folosită atât de client, cât și de server.
- **Logging** (`logging.h & logging.cpp`)
  Conține o funcție `log_message` care scrie mesaje cu timestamp într-un fișier `server.log`.
- **Makefile**
  Conține regulile de compilare (`g++`, `-Wall`, `-pthread`, `-lsqlite3`) și linkare pentru generarea binarelor `client` și `server`.

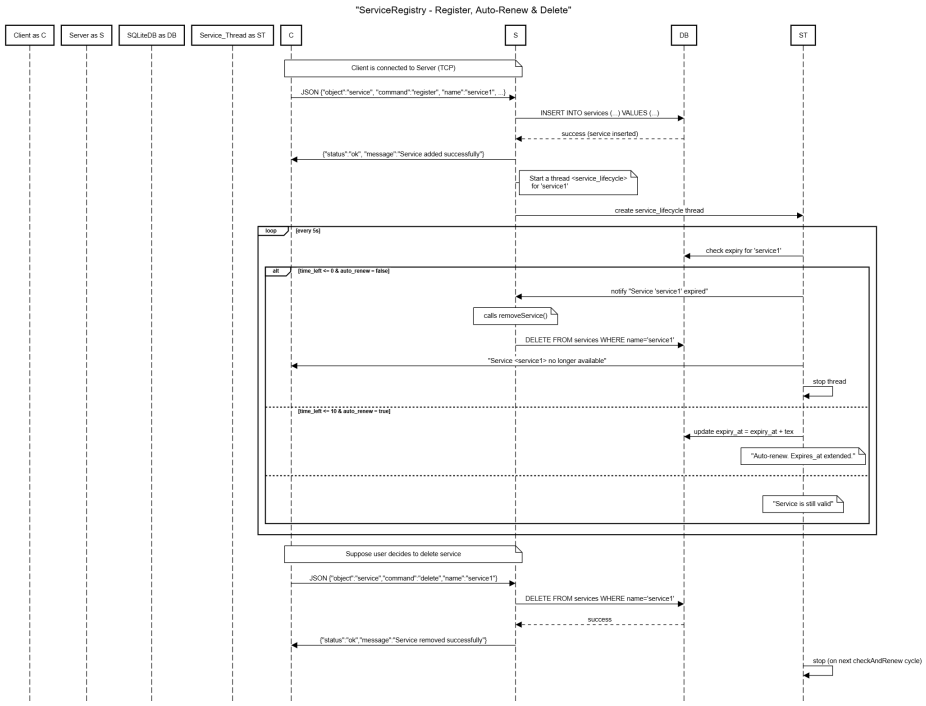
### 3.2 Mecanism de Înregistrare / Expirare Serviciu

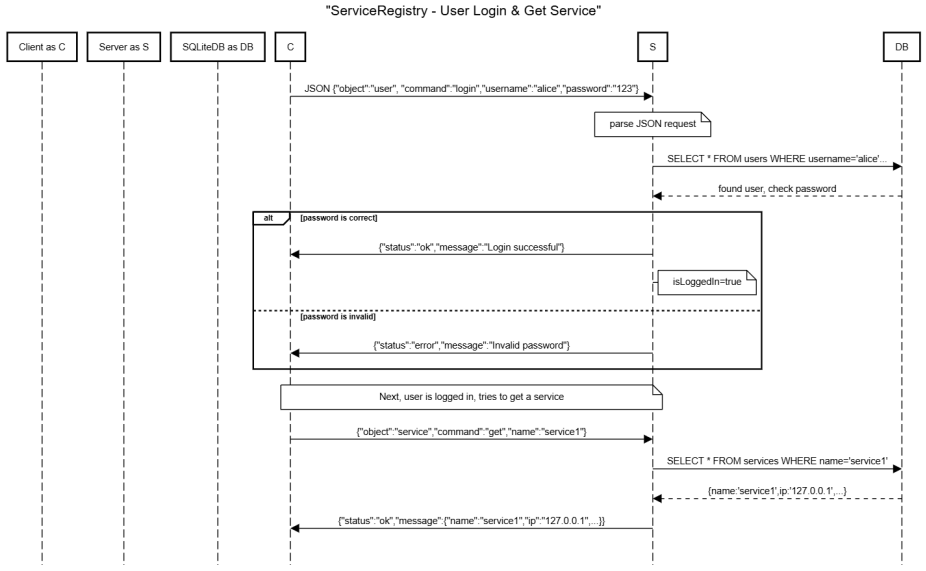
La **înregistrarea** unui serviciu, serverul inserează datele în tabela `services` (nume, IP, port, `TEx`, `auto_renew`) și pornește un **thread separat** ce scrie într-un fișier `service_<name .txt` informații despre rularea serviciului. Periodic (la 5 secunde), se face:

1. Calcul **time_left = expires_at - now**.
2. Dacă `time_left <= 0` **și** `auto_renew == false`, serviciul e *expirat* și se șterge.
3. Dacă `time_left <= 10` **și** `auto_renew == true`, se **reînnoiește** `expires_at`.

```
static void *service_lifecycle(void *arg)
{
    service_thread_data *data = (service_thread_data *)arg;
    ServiceHandler *handler = ServiceHandler::getInstance();

    // every 5 seconds:
    // 1) simulate service work by writing to a file
    // 2) check expiration/renew
    while (true)
    {
        pthread_mutex_lock(&data->lock);
        bool stop = data->stop;
        std::string name = data->name;
        pthread_mutex_unlock(&data->lock);

        std::ofstream outfile("service_" + name + ".txt", std::ios::app);
        time_t now = time(NULL);

        if (stop){
            outfile << "Service " << name << " stopped at " << ctime(&now);
            break;
        }

        // simulate service work
        outfile << "Service " << name << " is running at " << ctime(&now);

        // check expiration/renew
        handler->checkAndRenew(name);

        sleep(5);
    }
    delete data;
    return NULL;
}
```

**Fig. 3.** Cod din `server.cpp` - funcția `service_lifecycle`.


"ServiceRegistry - Register, Auto-Renew & Delete"
Client as C
Server as S
SQLiteDB as DB
Service_Thread as ST
C
S
DB
ST
Client is connected to Server (TCP)
JSON {"object":"service", "command":"register", "name":"service1",...}
INSERT INTO services (...) VALUES (...)
success (service inserted)
{"status":"ok", "message":"Service added successfully"}
Start a thread <service_lifecycle> for 'service1'
create service_lifecycle thread
loop
[every 5s]
check expiry for 'service1'
alt
[time_left <= 0 & auto_renew = false]
notify "Service 'service1' expired"
calls removeService()
DELETE FROM services WHERE name='service1'
"Service <service1> no longer available"
stop thread
[time_left <= 10 & auto_renew = true]
update expiry_at = expiry_at + ten
"Auto-renew. Expires_at extended."
"Service is still valid"
Suppose user decides to delete service
JSON {"object":"service","command":"delete","name":"service1"}
DELETE FROM services WHERE name='service1'
success
{"status":"ok","message":"Service removed successfully"}
stop (on next checkAndRenew cycle)


"ServiceRegistry - User Login & Get Service"
Client as C
Server as S
SQLiteDB as DB
C
S
DB
JSON {"object":"user", "command":"login","username":"alice","password":"123"}
parse JSON request
SELECT * FROM users WHERE username='alice'...
found user, check password
alt
[password is correct]
{"status":"ok","message":"Login successful"}
isLoggedIn=true
[password is invalid]
{"status":"error","message":"Invalid password"}
Next, user is logged in, tries to get a service
{"object":"service","command":"get","name":"service1"}
SELECT * FROM services WHERE name='service1'
{name:'service1',ip:'127.0.0.1',...}
{"status":"ok","message":{"name":"service1","ip":"127.0.0.1",...}}

# 4 Baza de Date

Proiectul folosește o bază de date **SQLite** pentru stocarea persistentă a informațiilor despre utilizatori și servicii. Schema de bază include două tabele principale:

- **users**: *(id, username, password, role)*
  Este folosit pentru autentificare și gestionarea drepturilor de acces.
- **services**: *(id, name, ip, port, tex, auto_renew, expires_at, active, last_update)*
  Conține toate datele despre serviciile înregistrate în aplicație.

| Name | Type | Schema |
|---|---|---|
| **services** | | CREATE TABLE services (id INTEGER PRIMARY KEY AUTOINCREMENT,name TEXT NOT NULL UNIQUE,ip TEXT NOT NULL,port INTEGER NOT NULL,last_update TEXT NOT NULL,tex INTEGER NOT NULL,auto_renew INTEGER NOT NULL DEFAULT 0,expires_at INTEGER NOT NULL,active INTEGER NOT NULL DEFAULT 1) |
| id | INTEGER | "id" INTEGER |
| name | TEXT | "name" TEXT NOT NULL UNIQUE |
| ip | TEXT | "ip" TEXT NOT NULL |
| port | INTEGER | "port" INTEGER NOT NULL |
| last_update | TEXT | "last_update" TEXT NOT NULL |
| tex | INTEGER | "tex" INTEGER NOT NULL |
| auto_renew | INTEGER | "auto_renew" INTEGER NOT NULL DEFAULT 0 |
| expires_at | INTEGER | "expires_at" INTEGER NOT NULL |
| active | INTEGER | "active" INTEGER NOT NULL DEFAULT 1 |
| **sqlite_sequence** | | CREATE TABLE sqlite_sequence(name,seq) |
| name | | "name" |
| seq | | "seq" |
| **users** | | CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT,username TEXT NOT NULL UNIQUE,password TEXT NOT NULL,role TEXT NOT NULL) |
| id | INTEGER | "id" INTEGER |
| username | TEXT | "username" TEXT NOT NULL UNIQUE |
| password | TEXT | "password" TEXT NOT NULL |
| role | TEXT | "role" TEXT NOT NULL |

**Fig. 4.** Schema tabelelor din SQLite (`users` și `services`).

Pentru orice inserare, ștergere sau actualizare, aplicația apelează metodele definite în `UserRepository` și `ServiceRepository`, care gestionează direct interogările SQL (ex.: `INSERT`, `UPDATE`, `DELETE`, `SELECT`).

# 5 Aspecte de Implementare

## 5.1 Autentificare și Controlul Accesului

Orice nou client trebuie mai întâi să fie **logat** (fie `login`, fie `register`), altfel operațiile asupra serviciilor sunt refuzate cu `"Please log in or register first"`. De exemplu, `"object": "user", "command": "login"` primește `"username"`

și `"password"` și caută în **users** (SQLite) corespondența. Dacă validarea reușește, `td->isLoggedIn = true` pentru thread-ul curent.

```
if (!td->isLoggedIn)
{
    if (object == "user")
    {
        if (command == "register")
        {
            std::string username = parser["username"].getString();
            std::string password = parser["password"].getString();
            std::string role = parser["role"].getString();

            User newUser(username, password, role);
            return userHandler->addUser(newUser);
        }
        else if (command == "login")
        {
            std::string username = parser["username"].getString();
            std::string password = parser["password"].getString();
            Response res = userHandler->login(username, password);
            if (res.status == "ok")
            {
                td->isLoggedIn = true;
            }
            return res;
        }
    }
    return Response{"error", "Please log in or register first"};
}
else
{
    if (object == "user" && command == "logout")
    {
        td->isLoggedIn = false;
        return Response{"ok", "Logged out successfully"};
    }
```

**Fig. 5.** Fragment `processRequest` pentru `login` din `server.cpp`.

### 5.2 Repository Pattern

Pentru gestionarea entităților **User** și **Service** din baza de date, sunt definite clase de tip *Singleton Repository*:

- **UserRepository**: creează și inițializează tabela *users* (ID, username, password, role), metode: `add`, `getByUsername`, etc.
- **ServiceRepository**: creează și inițializează tabela *services* (ID, name, ip, port, tex, expires_at, etc.), metode: `add`, `remove`, `update`, `getAll`, `getByName`.

Astfel, logica SQL este izolată de restul serverului.

### 5.3 Logare și Fișiere Auxiliare

Fiecare cerere și răspuns se **loghează** într-un fișier `server.log`, prefixat cu timestamp. În plus, pentru fiecare serviciu nou se creează un fișier `service_<name .txt`, unde thread-ul dedicat acelui serviciu scrie la fiecare 5 secunde un mesaj *"Service X is running at ..."* și la oprire *"Service X stopped at ..."*.

```
void log_message(string message) {
    time_t now = time(nullptr) ;
    if (message[message.length() - 1] == '\n')
        message = message.substr(0, message.length() - 1);
    // will add the timestamp to the log message in the format [YYYY-MM-DD HH:MM:SS]
    log_file << "[" << put_time(localtime(&now), "%Y-%m-%d %H:%M:%S") << "] " << message << endl;
}
```

**Fig. 6.** Fragment de cod din `logging.cpp`, funcția `log_message`.

## 6 Concluzii

Proiectul *Service Registry* demonstrează o aplicare practică a conceptelor de **rețelistică**, **arhitectură distribuită** și **gestionare a resurselor**. Prin utilizarea unui server concurent bazat pe `pthread`, comunicarea fiabilă prin protocolul **TCP**, și implementarea unui *Repository Pattern* pentru operarea bazei de date **SQLite**, am construit o soluție pentru descoperirea și administrarea serviciilor într-o rețea:

- **Protocol TCP** și mesaje **JSON** pentru schimb fiabil de date.
- **Server concurent**, folosind **thread**-uri pentru fiecare client și lifecycle threads pentru servicii.
- **Autentificare** și **Repository Pattern** pentru gestiunea entităților (utilizatori, servicii) în **SQLite**.
- **Logarea** cererilor și simularea activității fiecărui serviciu în fișiere dedicate.

Pe lângă funcționalitățile de bază, cum ar fi **înregistrarea**, **actualizarea** și **listarea serviciilor**, proiectul aduce elemente suplimentare precum **autentificarea utilizatorilor**, **logarea activităților** și mecanisme automate de reînnoire și expirare a serviciilor. Aceste caracteristici contribuie la creșterea fiabilității și securității aplicației, oferind un punct de plecare solid pentru extinderi ulterioare.

Printre direcțiile viitoare de dezvoltare la care mă gândesc se numără integrarea unui sistem de criptare pentru datele sensibile transmise între client și server, optimizarea performanței serverului pentru a suporta un număr mai mare de conexiuni simultane, precum și extinderea funcționalităților API-ului pentru a include metrici de performanță și analize statistice asupra serviciilor gestionate.

## 7 Referințe Biografice


[1] Pagina web Springer LNCS Guidelines, utilizată pentru a elabora raportul tehnic cu formatul specificat, cât și pagina utilizată pentru redactarea în LaTeX:
https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines
https://www.overleaf.com/project

[2] Site-ul pe care l-am utilizat pentru a realiza diagrama arhitecturii aplicației:
https://sequencediagram.org

[3] Site-ul materiei Rețele de Calculatoare, de unde am luat majoritatea informațiilor:
https://edu.info.uaic.ro/computer-networks/

[5] Site-uri utilizate pentru informații în legătură cu bazele de date SQLite (cum să le folosesc, tutoriale, modul de funcționare, exemple):
https://www.sqlite.org/index.html https://www.sqlitetutorial.net/
https://www.tutorialspoint.com/sqlite/index.htm

[6] Site-uri pe care le-am folosit pentru detalii suplimentare despre TCP:
https://en.wikipedia.org/wiki/Transmission_Control_Protocol
https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/
https://www.freecodecamp.org/news/tcp-vs-udp/

[8] Libpthread POSIX – https://man7.org/linux/man-pages/man7/pthreads.7.html

[9] Surse adiționale:
https://www.youtube.com/
https://stackoverflow.com/
https://en.cppreference.com/w/