Collaborative Notepad

Chiriac Laura Florina

Universitatea Alexandru Ioan Cuza, Facultatea de Informatică, Iași

https://www.info.uaic.ro/

Abstract. Raport tehnic pentru aplicație client/server de editare simultană a fișierelor text.
Keywords: TCP concurent · thread · bază de date relațională.

1.Introducere

Proiectul se bazează pe implementarea unui client care se conectează la server (la care au acces mai mulți utilizatori, dar cu limitarea lor pe un document) și poate trimite comenzi în urma cărora, serverul, trimite un mesaj de confirmare a primirii și face operațiile pe care le dorește utilizatorul. De asemenea, utilizatorul poate interacționa cu un editor de text în cadrul clientului, iar serverul va primi și procesa automat acel conținut, realizându-se sincronizarea.

"Collaborative Notepad" reprezintă o aplicație de tip server/client ce oferă posibilitatea utilizatorilor înregistrați să înceapă sesiuni concurente de editare text pe documente diferite, în pereche cu maxim încă un utilizator pe fiecare document. Acesta va putea transmite (prin intermediul unei interfețe grafice) comenzi de editare text, comenzi legate de fișierele și folderele stocate pe server, comenzi care țin de sesiunea de conectare:

A. Comenzi de manipulare a fișierelor și folderelor stocate pe server (ca elemente abstracte):

- Comenzi pentru lucrul cu fișiere: *create_document, edit, find_document, save_document, delete_document, download_document, modify_docdata* (se poate modifica doar numele sau locația – precizare prin parametru)
- Comenzi pentru lucrul cu directoare: *create_folder, delete_folder, find_folder, modify_folderdata* (se poate modifica doar numele sau locația – precizare prin parametru)

B. Comenzi pentru editare text: *edit* (comanda de început pentru editare text), în timpul editării : comenzi de tipul *modify_text* cu parametri pentru adăugare text, ștergere text, selectare text, copiere text, lipire text, modificare stil text.

C. Comenzi de gestionare a sesiunii de lucru: *login, logout, sign-up, close session*.

2.Tehnologii Aplicate

Pentru realizarea proiectului am recurs la folosirea unui server TCP concurent. TCP este un protocol folosit în aplicații care au nevoie de siguranța confirmării primirii datelor, de păstrarea ordinii, de stabilitatea menținerii unei conexiuni între expeditor și receptor. Am folosit acest protocol deoarece în aplicația mea am nevoie de o conexiune stabilă și de păstrarea ordinii mai ales, deoarece contează păstrarea ordinii operațiilor peste documentele text din server. Mai mult, am folosit modelul concurent și nu de cel iterativ deoarece aplicația trebuie să permită conectarea mai multor clienți în același timp cărora să le răspundă prin prisma modelului de colaborare între utilizatori, care iau parte la sesiuni concurente de operare asupra fișierelor. În ciuda faptului că proiectul meu necesită o transmitere rapidă a informației, nu am considerat potrivit un server UDP, deoarece o alterare sau o pierdere a informației în cadrul aplicației ar putea un efect fatal asupra datelor cu care se lucrează dar și asupra mediului de lucru din aplicație. [1]
Pentru asigurarea concurenței am folosit thread-uri, întrucât este o metodă rapidă și eficientă de a diviza procesele pentru clienți. Fiind independente unul de celălalt, asta permite unui client să facă cereri către server independent de acțiunile unui alt client. [3]
În ceea ce privește bazele de date folosite, am folosit baze de date separate pentru metadatele documentelor și pentru datele despre fiecare user în parte.

Pentru arborele de fișiere care aparțin fiecărui utilizator am ales să folosesc un fișier xml. Fișierul conține date despre toate documentele stocate pe server, care pot fi accesate de anumiți utilizatori. Am folosit un astfel de fișier deoarece este potrivit pentru a descrie un arbore pe mai multe nivele cu o organizare a unor fișiere. Am folosit tag-uri pentru fiecare document al unui user de tipul <title>, <file_id>, <location>, <created>, <last_modified> - care descriu atributele standard ale unui document. Trebuie precizat că *file_id* este identificatorul unic al unui fișier, iar locația care este setată in mod implicit este *home_dir* - reprezintă un folder principal al fiecărui user, în care sunt stocate la creere toate fișierele și restul folderelor. Denumirea fiecărui fișier va fi stabilită la salvare chiar de utilizator, care poate solicita să schimbe și locația standard.

Aplicația dispune și de o bază de date a utilizatorilor. Aceasta asigură, pe de o parte, securitatea datelor unui utilizator, deoarece nu oricine poate face operații pe fișierele de pe server așa cum dorește, dar ajută și la identificarea conectărilor și deconectărilor atât de pe server, cât și conectarea la sesiunile de editare ale fișierelor. Va ajuta în implementare și la identificarea utilizatorilor care sunt activi pe un anumit document sau dintr-o anumită sesiune, pentru

a putea identifica modalitatea în care trebuie serverul să gestioneze situațiile critice (de exemplu, când un al treilea client încearcă să acceseze pentru editare un fișier deja editat în timp real de o altă pereche).

3. Structura Aplicației

3.1 Conceptele folosite în modelare

În modelarea aplicației, am folosit ca și concepte server, client, bază de date (relațională), fișier xml. Serverul primește cereri de conectare de la mai mulți clienți simultan și răspunde acestora pe baza concurenței, tot simultan. Fiecare client va trimite anumite comenzi – cereri de a efectua anumite operații asupra fișierelor sau a mediului de editare, iar în funcție de comanda trimisă, serverul va răspune atât prin trimiterea unui mesaj de confirmare cât și, cel mai important, prin efectuarea operațiilor cerute pe baza sa stocată de date. De altfel, se va transmite și înapoi un semnal de atenționare astfel încât să se sincronizeze și clientul cu schimbările efectuate pe server. Trebuie precizat că fără a fi logat, un utilizator nu are acces la comenzile de prelucrare a informațiilor, deci mai întâi serverul mereu verifică dacă utilizatorul ce folosește un client este logat într-un cont sau nu, folosindu-se de informațiile din tabelul de utilizitatori. Funcționarea serverului se bazează, astfel în primul rând pe bazele de date pe care le folosește pentru a identifica cum trebuie să se comporte atunci când efectueaza operațiile solicitate de clienți.
Comunicarea cu baza de date pentru utilizatori se face cu ajutorul librăriei sqlite3.h. SQLITE este un sistem de gestiune a bazelor de date relaționale bazat pe limbajul SQL. Spre deosebire de majoritatea altor baze de date SQL, SQLite nu are un proces server separat, acesta fiind un avantaj major ce ține de eficiența proiectelor în care este el utilizat. Alegerea acestei librării vine și din ușurința cu care pot fi transformate informațiile stocate în baza de date, lucru foarte util deoarece modificările trebuie să se poată face destul de des în aplicația descrisă. De asemenea, modelul de baze de date relațional pe care îl descrie SQL este necesar pentru identificarea, de exemplu, a cheilor primare sau a celor străine, și pentru a face, bineînțeles, legătura între două sau mai multe relații (tabele, în limbaj informal), care împreună conțin noi informații importante despre întreg sistemul de date, necesar în dezvoltarea aplicației. [2]

Prima tabelă din baza de date descrisă anterior conține informațiile despre fiecare utilizator – ID – cheie primară, care va fi unic pentru fiecare utilizator, Nume – text care nu poate să fie NULL (și, la fel, diferit pentru fiecare utilizator), Parola, Status – 1 dacă utilizatorul e conectat și 0 dacă nu e conectat.

ID	Nume	Parola	Status
1	utilizator1	1234pass	0
2	utilizator2	admin311	1

Fig.1. *Tabela Utilizatori din baza de date db in SQLITE*

Cealaltă tabelă din baza de date descrisă anterior conține informațiile despre fiecare Document – cele legate de utilizatori și nu detaliile despre fiecare document în parte. Aceasta face legătura dintre sesiunile de conectare ale clienților pe care lucrează fiecare dintre utilizatori și documente, precum gestionează și păstrarea conținutui scris de documente, care va fi modificat constant prin intermediul editorului de text. Totuși, trebuie menționat că informațiile se stochează de abia după salvare, când documentul este practic unul în sine. Dacă utilizatorul nu le salvează, acestea vor fi pierdute. Totodată, se vor putea salva și documente fără conținut, dar care au setat un nume, o dată la care a fost creat, etc., toate fiind la început fără conținut, de fapt. Pentru a exemplifica modul în care doresc să utilizez baza de date în proiect, am introdus câteva date și pentru Documente care vor indica funcționalitatea ei în aplicație.

ID	ID_Utilizator	Continut
1	1	Acesta e un text scris de utilizatori in document.
2	1	Acesta e al doilea text pentru testare.
3	2	Text adaugat de un alt user, eventual impreuna cu un alt user.

Fig.2. *Tabela Documente din baza de date db in SQLITE*

Tabela Documente este o tabelă importantă pentru întregul proiect, deoarece ajută și la gestionarea celorlalte baze de date existente. Aceasta conține chei care indică atât un id-ul pentru fiecare utilizator, cât și cel unic al unui document. Astfel, se realizează o legătură între cele 2 și text, care poate ajuta la identificarea celor care editează textul la

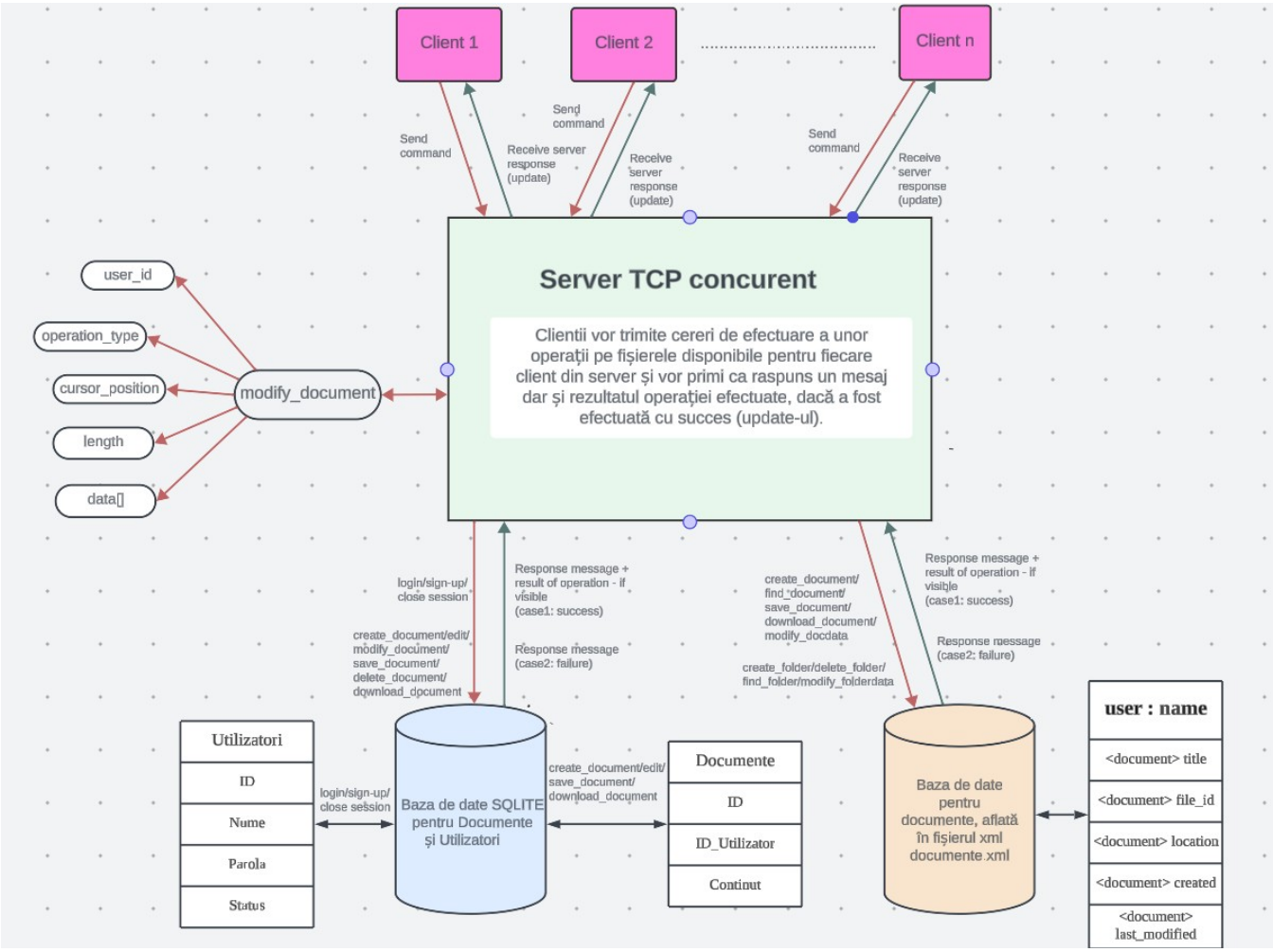
un moment dat (de exemplu, dacă există de două ori introdus în tabelă același ID, înseamnă că acei utilizatori cu id-uri de utilizator ID_Utilizator corespunzătoare editează acel fișier în timp real. Diferența între acest câmp al tabelei Documente și tag-ul din fișierul xml (despre care voi vorbi mai mult în ceea ce urmează) este aceea că tag-ul reprezintă utilizatorul care a creat documentul în primă fază - autorul, dar aici voi avea id-ul autorului care editează conținutul din fișier.

Pentru a stoca informațiile despre fiecare document în parte, am folosit și un fișiser xml care descrie metadatele unui fișier obișnuit : când a fost creat, modificat ultima dată, locația sa (adică folderul în care se află), titlul care trebuie neapărat să existe. Pentru a integra fișierul xml în proiect, am inclus bibliotecile ”libxml/parser.h” și “limxml/tree.h”, care ajută la parsarea în C a conținutului unui fișier xml și a tag-urilor folosite, dar care ajută și la descrierea structurilor dintr-un arbore rezultat din parsarea fișierului xml și să accesez sau să modific acest conținut. Dintre operațiile pe care le-aș putea face asupra acestui fișier ar fi, de exemplu, modificarea titlului unui document când primesc o cerere corespunzătoare de la client, sau să modific data ultimei modificări atunci când un utilizator termină de modificat fișierul respectiv, ori să caut efectiv un document – atunci când mi se cere acest lucru de către un utilizator care rulează comanda ”find_document” urmat, ca paramtetru, de un identificator al documentului.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<documentsList>
    <user name="utilizator1">
        <document>
            <title>Untitled1</title>
            <file_id>1</file_id>
            <location>home_dir</location>
            <created>11.12.2023 12:13</created>
            <last_modified>11.12.2023 14:11</last_modified>
        </document>
        <document>
            <title>Untitled2</title>
            <file_id>2</file_id>
            <location>home_dir</location>
            <created>12.12.2023 14:01</created>
            <last_modified>12.12.2023 14:25</last_modified>
        </document>
    </user>
    <user name="utilizator2">
        <document>
            <title>Untitled1</title>
            <file_id>3</file_id>
            <location>home_dir</location>
            <created>12.12.2023 04:01</created>
            <last_modified>12.12.2023 21:20</last_modified>
        </document>
    </user>
</documentsList>
```

Fig.3. *Fișierul xml documente.xml care reține structura documentelor stocate și datele despre acestea*

3.2 Diagrama detaliată a aplicației



4. Aspecte de Implementare

4.1 Protocol de comunicare

Protocolul de comunicare pe care l-am stabilit pentru această aplicație este unul destul de complex, deoarece prin intermediul acestuia se gestionează mai multe tipuri de operații care pot fi transmise de mai mulți clienți în același timp la server și pot fi chiar în conflict.

În ceea ce privește operațiile principale pe fișiere și cele similare pe directoarele în care sunt grupate acestea, avem:
Crearea unui nou fișier: doar se transmite comanda, se recunoaște în server, și apoi se efectuează modificarea în baza de date și se trimite răspunsul la client. În acest punct, se va "activa" o stare de "editing" în server, iar la această activare se va notifica și clientul (această parte va fi utilă mai ales când se va adăuga o interfață proiectului, în care în acest punct de început de editare va trebui să se activeze și o fereastră de editare text, care să îi permită utilizatorului să introducă textul).

Descărcarea unui fișier existent: se primește în server comanda de la un client, se va căuta în server cu aceeași metodă de la *find_document* și se va începe descărcarea. Din nou, va fi notificat și clientul, iar apoi se vor trimite datele necesare, pentru a putea fi puse într-un fișier concret (adică care nu mai e abstract sau reprezentat printr-o bază/structură de date). O idee de implementare a acestui protocol este preluarea datelor și trimiterea lor pe rând : de exemplu, mai întâi trimitem datele despre fișier propriu-zis (extrase prin parsing din fișierul cu informații despre document, dar și din tabelă prin SELECT), apoi se trimite conținutul, acestea sub forma unui mesaj cu o structură șablon, stabilită, în care poate fi recunoscut conținutul și reconstituit totul. La reconstrucție tot ce se primește ca și conținut va fi scris (cu write; ca intr-un fișier de output) în fișierul cu datele specificate (datele sunt în primul mesaj transmis la download). Astfel, vom avea

reconstituit fișierul virtual într-un fișier real, iar în acest sens se poate adăuga un folder local pentru fiecare client sau pentru fiecare utilizator în parte.

Modificarea textului în fișier:

În client: Se va primi comanda, va fi transmisă comanda la server, și, împreună cu această comandă care trebuie să specifice operația exactă - ștergere caractere (din editor), lipire, adăugare de la tastatură – va fi transmisă și o structură de date corespondentă operației de modificare a textului (*modify_text*), care va conține date despre cine a făcut modificarea, tipul modificării, poziția cursorului celui care a modificat, dimensiunea modificării (număr de caractere) și ce alte date mai trebuie transmise în funcție de tipul operației. Transmiterea trebuie să se facă în acest fel tipizat încă de la client, deoarece clientul are acces facil la aceste date. Și în server se va implementa această structură, care va fi, la fel, populată cu datele primite de la client, sub o formă de mesaj în care s-ar putea transmite fără pierderi și greșeli.

În server: Se primește în server comanda transmisă de client, structura modificării, și după ce e stabilită natura operației, se aplică operația corespunzătoare. Deoarece operația aceasta trebuie să se desfășoare repede, nu mai e nevoie de fiecare dată să se trimită mesaj de confirmare. Atunci când proiectul are și interfață, mai ales, maxim se poate afișa un mesaj de *Editing…* care să rămână acolo, pentru a fi evident ce operație se efectuează. În schimb, se va modifica conținutul documentului în mod corespunzător, pentru fiecare dintre operații. Înapoi la client, va fi transmisă modificarea pentru a fi vizibilă vizual utilizatorului, sau mai simplu – se poate sincroniza pur și simplu fișierul de fiecare dată, deoarece el pe server este deja modificat în acest punct, nemaifiind nevoie decât să se transmită clientului că a fost o modificare făcută cu succes.

```
struct modify_text {
    int user_id;
    int operation_type;
    int cursor_position;
    int length;
    char data[500];
}
```

Fig.4. *structura corespunzătoare operației de modificare a textului*

```
/*

Exemplu de mesaj codificat, corespunzator modify_text :
mesaj_modify = [user_id][operation_type][cursor_position][length][data (text mai
mare, eventual cu o codificare pentru diferite formatari posibile)];

*/
```

Fig.5. *Exemplu de mesaj codicat pentru trimitere - special pentru comanda modify_text*

Salvarea fișierului pentru a fi stocat pe server: se va primi comanda de la client care va indica faptul că trebuie să se efectueze salvarea și în server se va revizui baza de date ce ține de documentul respectiv, iar dacă este prima salvare se va prelua informația, care a fost reținută de program într-un buffer până în acest punct, și se va completa în baza de date conținutul fișierului. Către client, poate fi transmis înapoi, la finalul acestei operațiuni, un mesaj care să confirme sau nu salvarea fișierului text.

În ceea ce privește operațiile ce țin de sesiunile de lucru, avem:

Crearea și închiderea sesiunilor: o sesiune de editare pe un singur document trebuie să fie partajată de cel mult 2 clienți. Astfel, o nouă sesiune de editare va începe atunci când se intră în starea de *"Editing"* pe un fișier sau mai multe. La această sesiune se poate conecta și un alt utilizator, dacă încearcă să editeze fișierul cu același *file_id*. Însă, dacă de la un al treilea client se primește o cerere similară, ea va fi respinsă pe moment, deci serverul va returna un mesaj de eroare. De asemenea, serverul ar trebui să notifice toți utilizatorii dacă unul dintre cei aflați pe fișierul respectiv, de exemplu, s-a deconectat, s-a apucat de editat, etc. Așadar, odată ce doi clienți editează pe un același document de pe server, având ambii pornită câte o sesiune de editare în care se editează și acel document, serverul trebuie să fie pregătit să notifice pe toți utilizatorii prezenți în acest mediu de editare de operațiile pe care le efectuează/doresc să le efectueze ceilalți utilizatori, din clientul lor. Notificările vor fi transmise folosind o formă specială a mesajului, de exemplu un

keyword trimis odată cu mesajul notificării, sau tot o formă cu paranteze (exemplu: (N)textul_notificarii), pentru a ști că trebuie să fie vizibile explicit pentru utilizatorul colaborator din sesiunea comună de editare a fișierului.

Când un utilizator se deconectează, sau pur și simplu închide un fișier care e în colaborare cu un alt utilizator, serverul va trebui să oprească transmiterea de notificări despre acțiunile celuilalt către acesta. Și invers acest lucru este valabil (pentru utilizatorul care este activ în editor), ultima notificare pe care o va primi acesta fiind totuși cea că celălalt a făcut request să se deconecteze de la sesiunea de editare pe document. Astfel, se va opri conexiunea dintre cei doi pentru moment.

Când iese ultimul utilizator activ de pe un fișier, se va efectua închiderea. Fiindcă nu mai e și altcineva care să colaboreze cu el la editarea fișierului pe moment, pur și simplu se stochează datele modificate și se modifică data ultimei modifări, fără a trebui o notificare suplimentară sau orice altceva.

Trebuie precizat, în plus, că la fiecare dintre aceste operațiuni vom avea răspunsuri diferite în funcție de anumite cazuri, aspect pe care îl vom discuta în cadrul secțiunii următoare, cu scenariile de utilizare.

4.2 Scenarii de utilizare

Pentru început, trebuie menționat faptul că toate funcțiile sunt neutilizabile (serverul va transmite un mesaj de eroare) atâta timp cât nu este vreun utilizator logat pe clientul respectiv (lucru contorizat de variabila globală logged_in).

Așadar, în ceea ce privește operațiile principale pe fișiere și cele similare pe directoarele în care sunt grupate acestea, avem ca scenarii de utilizare diferite de cele care nu impun probleme următoarele:

Crearea unui nou fișier: la crearea unui fișier, singurul scenariu negativ este ca un fișier cu același nume ca altul deja existent să fie creat.

Descărcarea unui fișier existent: la descărcarea unui fișier, există posibilitatea ca el să nu existe. Acest caz se rezolvă doar trimițând un mesaj înapoi (de eroare).

Modificarea textului în fișier: modificarea textului în fișier poate impune mai multe probleme. În primul rând, se poate ca doi utilizatori de pe un fișier să încerce editarea aceleași porțiuni în același timp. Deoarece avem structura modify_text, vom gestiona cu ajutorul datelor de acolo acest lucru. Principiul de bază care se aplică este că se formează un fel de coadă a cererilor de editare în acel loc, ca serverul să execute operațiile în ordinea în care au fost primite de acesta, rezultat care poate fi totuși incert, dar care are sens în contextul mai larg. Dacă nu este de ajuns, problema se poate rezolva printr-o notificare a serverului către clienți, care să anunțe utilizatorii că încearcă să facă operații care sunt conflictuale. Astfel, dacă două operații sunt într-un conflict nerezolvabil, li se vor oferi opțiuni direct utilizatorilor de a gestiona potențialele conflicte. În acest caz, dacă chiar și opțiunile celor 2 intră în conflict, notificarea va persista până când există o hotărâre de comun acord (sau, mai concret, doar până unul dintre ei reușește într-un interval de timp mai scurt de la primirea notificării să aleagă opțiunea dorită).

Salvarea fișierului pentru a fi stocat pe server: singurul scenariu anormal sau negativ de utilizare este atunci când mai există un alt fișier cu același nume.

Mai mult, în ceea ce privește operațiile ce țin de sesiunile de lucru, avem următoarele scenarii;

Crearea și închiderea sesiunilor: crearea unei sesiuni poate impune probleme doar pentru un al treilea client, care face un request să editeze un document pe care lucrează numărul maxim de utilizatori. În acest caz, trebuie verificat documentul tot timpul la acceptarea unei conexiuni la sesiunea de editare. Închiderea, în schimb, trebuie menționat că se poate efectua fără restricții, atât că există pericolul să nu se salveze conținut.

Gestionarea mai multor sesiuni simultane mai este, în plus, o problemă ce poate apărea în utilizare, dar care se poate rezolva aplicând în continuare principiile de la “Editing” și crearea/închiderea de sesiuni de editare pe documente.

4.3 Secțiuni de cod cheie

În comunicarea dintre server și client am folosit socket, deoarece este un canal de comunicație bidirecțional, mult mai simplu de folosit și mai performant decât altele, în cadrul cărora sunt implementate separat citirea și scrierea. Totodată, dacă ne referim la contextul proiectului, socketul este utilizat în mod obișnuit pentru comunicarea în rețea, deoarece în rețea se transmite un volum mare de date și dintr-o parte și din alta a canalului. [4]

Pentru asigurarea concurenței am folosit thread-uri, iar astfel clientul poate să facă un număr nelimitat de request-uri către server, iar server-ul se va închide doar la tastarea comenzii close session.

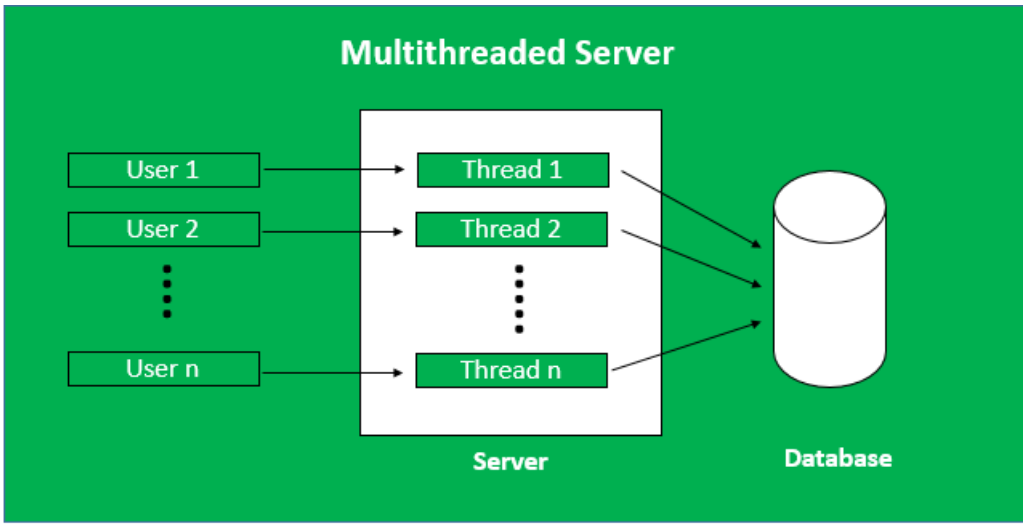


Fig.6. *Reprezentare grafică implementare cu multiple thread-uri separate*

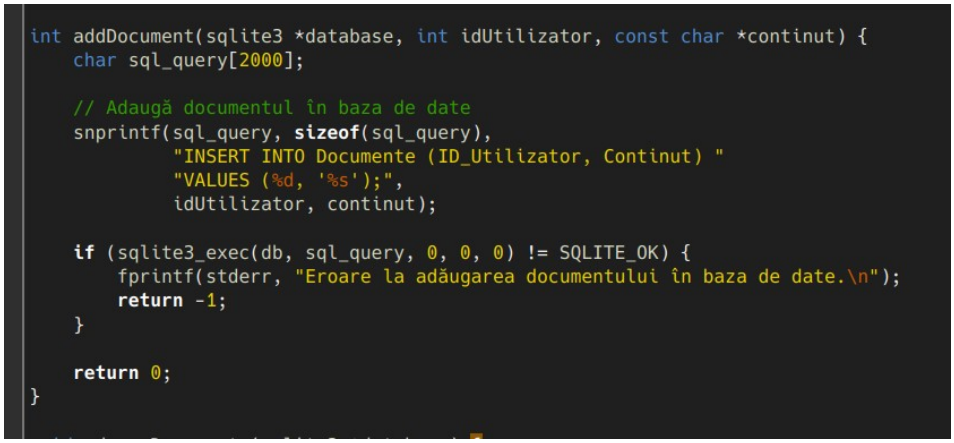
Ca secțiuni de cod cheie care nu au mai fost prezentate, dintre care se află deja implementate în aplicație, pot fi precizate funcțiile pentru operarea cu bazele de date.

- SqlDatabase() : care deschide baza de date, crează tabelele dacă nu există și permite accesul la baza de date pentru a prelua informațiile

```
int SqlDatabase() {
    //sqlite3 *db;
    char *error = 0;

    int rc = sqlite3_open("server_data.db", &db);
    if (rc) {
        fprintf(stderr, "Nu s-a putut deschide baza de date! %s\n",
        sqlite3_errmsg(db));
        return rc;
    }

     // Crearea tabelului "Utilizatori" dacă nu există deja
    const char *createUsersTableSQL =
        "CREATE TABLE IF NOT EXISTS Utilizatori ("
        "ID INTEGER PRIMARY KEY AUTOINCREMENT, "
        "Nume TEXT NOT NULL, "
        "Parola TEXT NOT NULL, "
        "Status INTEGER NOT NULL);";

    rc = sqlite3_exec(db, createUsersTableSQL, 0, 0, &error);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "Eroare la crearea tabelului Utilizatori! %s\n", error);
        sqlite3_free(error);
        sqlite3_close(db);
        return rc;
    }

    // Crearea tabelului "Documente" dacă nu există deja
    const char *createDocumentsTableSQL =
        "CREATE TABLE IF NOT EXISTS Documente ("
        "ID INTEGER PRIMARY KEY AUTOINCREMENT, "
        "ID_Utilizator INTEGER NOT NULL, "
        "Continut TEXT, "
        "FOREIGN KEY (ID_Utilizator) REFERENCES Utilizatori(ID));";

    rc = sqlite3_exec(db, createDocumentsTableSQL, 0, 0, &error);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Eroare la crearea tabelului Documente: %s\n", error);
        sqlite3_free(error);
        sqlite3_close(db);
        return rc;
    }
```

Fig.7. *Funcția SqlDatabase()*

- addUser : funcția care (va) permite adăugarea unui nou utilizator la baza de date
- addDocument : care adaugă un document la baza de date (din SQLITE)



```
int addDocument(sqlite3 *database, int idUtilizator, const char *continut) {
    char sql_query[2000];

    // Adaugă documentul în baza de date
    snprintf(sql_query, sizeof(sql_query),
             "INSERT INTO Documente (ID_Utilizator, Continut) "
             "VALUES (%d, '%s');",
             idUtilizator, continut);

    if (sqlite3_exec(db, sql_query, 0, 0, 0) != SQLITE_OK) {
        fprintf(stderr, "Eroare la adăugarea documentului în baza de date.\n");
        return -1;
    }

    return 0;
}
```

Fig.8. *Funcția addDocument*

- verifyUserStatus : funcția (va) permite verificarea statusului unui user, pentru a ajuta la procesul de login

Pentru fișierele xml, voi avea:

- XMLFileParsing : pentru parsarea datelor din fișierul XML și facilitarea accesului la ele
- extract_XML : pentru a extrage din fișierul XML un anumit câmp

Funcții speciale pentru cereri:

- Modify_text : funcție specială care să se ocupe de operațiunile ce țin de modificarea textului
- Modify_doc_data : funcție care va modifica datele unui document – adică va păstra și legătura dintre datele din bazele de date ale aplicației.

Precizez că acestea sunt doar o parte din funcțiile utile pentru rezolvarea cererilor de la clienți, deoarece codul propriu-zis se mai poate modulariza/împărți în bucăți mai mici.

5. Concluzii

Proiectul se mai poate îmbunătăți considerabil. De exemplu, se poate reconstrui baza de date astfel încât accesarea informațiilor să se facă mai eficient, eficiența fiind importantă pentru o aplicație unde serverul primește foarte multe cereri și trebuie să acționeze foarte rapid.

Alte îmbunătățiri posibile, pot ține de adăugarea unor funcționalități. De exemplu, deoarece se pot descărca fișiere, consider că ar fi bine de implementat și o funcționalitate de upload a fișierelor, care după să poată fi editate în notepad-ul colaborativ.

O altă funcționalitate care ar putea fi adăugată ar fi cea de undo, redo și control al versiunilor. Asta ar presupune și stocarea unor versiuni anterioare ale fișierelor în bazele de date ale serverului, astfel încât să poată fi accesate de utilizator, în caz de nevoie. La fel, se poate gestiona și un model de undo al operațiilor. Adică, dacă s-a făcut o operație și a fost reținută, iar după s-a dat undo, atunci serverul ar putea să ”construiască” operația opusă și să transmită modificarea către clienți.

6. Bibliografie

1. https://www.geeksforgeeks.org/differences-between-tcp-and-udp/
2. https://www.sqlite.org/cintro.html
3. https://www.geeksforgeeks.org/multithreaded-servers-in-java/
4. https://www.ibm.com/docs/de/zos/2.1.0?topic=zos-communications-server