Vilcu Razvan-Valeriu
Grupa 2A5

StudyBuddy

Visual Study Group

1. Introducere

Proiectul acesta, alias "StudyBuddy", este o aplicatie realizata cu scopul comunicarii mai multor studenti(utilizatori) in timp real, de a stoca si partaja fisiere si de a permite studentilor(utilizatorilor) sa modifice anumite fisiere/proiecte in timp real.

Obiectivul acestuia este de a realiza o conexiune stabila intre utilizatori, permitand acestora sa faca schimb de informatii fara pierdere si de a le oferi un spatiul cat mai comod de lucru.

Functionalitatile proiectului cuprind:

1) Conectarea la serverul principal folosind o fereastra interactiva
2) Crearea si logarea unui client intr-un cont
3) Trimiterea de mesaje in timp real si stocarea acestora pentru revizualizare
4) Verificarea utilizatorilor conectati pe platforma simultan.
5) Incarcarea si descarcarea de fisiere dintr-un cloud accesibil doar de server
6) Utilizatorii pot modifica fisiere, incarca in cloud si extrage, creand un sistem de colaborare live cu alti utilizatori
7) Deconectarea automata la inchiderea aplicatiei

2. Tehnologii aplicate

In primul rand, conexiunea dintre utilizatori este realizata printr-un **protocol TCP concurent**, serverul creand un thread pentru fiecare utilizator nou conectat, deoarece informatiile primite sau trimise nu trebuie sa aiba pierderi si sa poata sa permita comunicarea intre utilizatori fara a fi nevoie de a astepta.

Pentru interfata grafica am folosit **libraria Grafica Qt**, deoarece prezinta o multitudine de caracteristici utile in a imi stiliza programul si de a realiza o interfata user-friendly.

Am ales pentru stocarea diverselor date *(utilizatori conectati, chat-ul, fisierele din cloud, utilizatori ce "vad" anumite fisiere* sa folosesc **JSON**, deoarece este usor de deserializat si citit continutul mai ales daca lucrez cu date ce trebuie transmise inapoi clientului

Am folosit **CMake** pentru a compila programul mai usor si pentru a introduce libraria Qt si dependecy-urile acesteia in compilarea acestuia.

3. Structura Aplicatie

Pentru a intelege mai bine structura aplicatiei o sa impart explicatiile in: **pre-login & post-login**

Pre-login:

Inainte de a se deschide aplicatia, fiecare utilizator va fi nevoit sa se conecteze la server. Conexiunea se va realiza printr-un protocol TCP concurent si fiecarui utilizator ii va fi atribuit un thread, ce primeste anumite comenzi de la client, asteptarea de mesaje de la client este realizat cu ajutorul unui ***select()***, pentru ca operatiile de tipul ***read()/write()*** sa nu fie blocante, deoarece ar bloca si aplicatia principala

Dupa conectarea *client-server* o noua fereastra urmeaza sa apara, unde utilizatorul poate sa isi creeze un cont cu care foloseasca aplicatia sau in cazul in care are deja un cont, poate sa se logheze cu acest cont. In cazul in care utilizatorul doreste sa isi creeze un cont nou, acesta va fi adaugat intr-un fisier *JSON*. Diferite mesaje de eroare vor aparea in functie de anumite erori din partea utilizatorului *(ex: uitarea parolei, la register cele doua parole nu coincid, etc.)*

Post-login:

Observam in fereastra principala trei butoane: **Chat Room, Files & Live working**, ce reprezinta 3 meniuri asociate functionalitatilor principale ale proiectului.

Chat Room: Aici utilizatorii, pot vedea toti ceilalti utilizatori conectati pe aplicatie, indiferent pe care meniu sunt si de asemenea pot transmite mesaje catre toti ceilalti utilizatori. Mesajele sunt stocate intr-un fisier *JSON* si folosind un **QTimer** un mesaj este trimis catre server, iar serverul transmite inapoi toate mesajele si utilizatorii care le-au scris, pentru a putea realiza comunicarea in timp real.

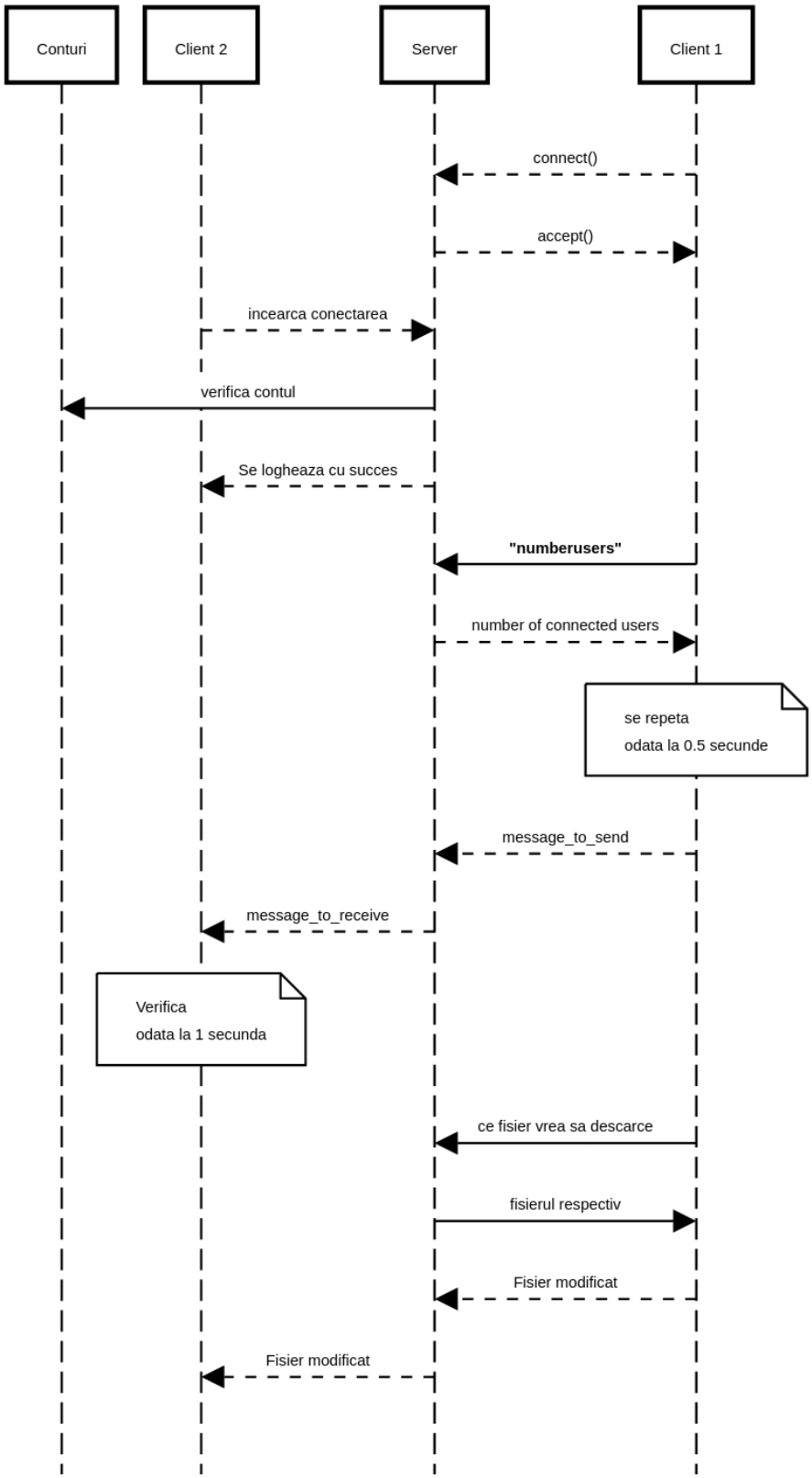
Files: In acest meniu, inca odata ne vom folosii de un **QTimer** pentru a verifica cate fisiere se afla in cloud, accesibile utilizatorilor. Observam si doua butoane, unul de download, iar altul de upload, unde utilizatorii pot descarca fisiere deja existente in cloud-ul aplicatiei sau pot incarcar fisiere din propriul calculator pentru a le putea partaja cu ceilalti utilizatori. Fisierele sunt incarcate si descarcate in mod binar, iar din cauza limitei impuse de **write()**, pentru a descarca fisiere mai mare, fisierele sunt sparte in bucati de 1024 bytes si citite in fisier tot pe bucati.

Live Sharing: Aceasta fereastra prezinta trei mici tabele, unul pentru fisierele din cloud, unul pentru utilizatorii conectati pe un anumit fisier si unul in care toti utilizatorii

sunt anuntati de modificarile facute de un alt utilizator. La fiecare click pe un anumit fisier, utilizatorul este inregistrat intr-un fisier de tip *JSON*. Partea de transmitere de fisiere este bazata pe transmiterea de fisiere realizata in meniul "Files", diferenta este ca atunci cand utilizatorul apasa pe butonul de *sync* fisierul este comparat cu cel din cloud, deoarece nu dorim abuzul butonului de sync, iar doar daca fisierele sunt diferite se va trimite un mesaj la toti utilizatorii care se uita la acelasi fisier un mesaj ca utilizatorul care a dat sync a si modificat fisierul.

Datorita functionalitatilor din libraria grafica *Qt*, se poate verifica cu usurinta cand un utilizator inchide aplicatia. Daca un utilizator inchide fereastra principala, un mesaj este trimis serverului pentru a modifica fisierul *JSON* notand ca utilizatorul este deconectat!

Structura aplicatiei "StudyBuddy"



4. Aspecte de Implementare

Protocolul pe care am ales sa il folosesc este TCP concurent, iar conexiunea va fi realizata dupa verificarea textului introdus in fereastra de inceput. Mai jos prezint cum este realizata conexiunea. Bucata de cod din fisierul `client.cpp` este preluat dinauntrul functiei care se declanseaza atunci cand este apasat butonul "Connect", de asta exista variabile de tip `QString`.

client.cpp

```
    QString server_address_temp = lineEdit_address->text();
    QString port_temp = lineEdit_port->text();

    if(server_address_temp.isEmpty() || port_temp.isEmpty())
    {
        errorConnect->setText("Completati␣ambele␣campuri!");
    }

    string server_address = server_address_temp.toStdString();
    string portString = port_temp.toStdString();

    port = 0;
    for(auto i : portString)
    {
        port = port*10 + (i - '0');
    }

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        errorConnect->setText("Eroare␣la␣socket()!");
        return errno;
    }

    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = inet_addr(server_address.c_str());

if (connect(sd, (struct sockaddr *)&server, sizeof(struct sockaddr)) ==
    -1)
    {
        errorConnect->setText("Eroare␣la␣connect()!");
        return errno;
    }
```

server.cpp

```
if(select(nfds+1, &readfds, NULL, NULL, &tv) < 0)
        {
            perror("Eroare la select()\n");
            return errno;
        }

        if(FD_ISSET(sd, &readfds))
        {
            len = sizeof(from);
            bzero(&from, sizeof(from));
            client = accept(sd, (struct sockaddr *)&from, &len);
            if(client < 0)
            {
                perror("Eroare la accept()");
                continue;
            }
            printf("S-a connectat clientul cu descriptorul: %d", client
               );
            FD_SET(client, &actfds);
            users_connected.push_back(client);
                fflush (stdout);
            if(nfds < client)
                nfds = client;
            printf("Clients count: %d\n", users_connected.size());

        }
```

Mai sus observam un array *users_connected*, am creat acest `vector<int>` pentru a tine minte descriptorii clientilor, pentru a imi fi mai usor sa transmit anumite informatii sau sa verific daca cineva s-a deconectat. Pentru ca folosesc thread-uri pentru a putea sincroniza utilizatorii conectati si descriptorii lor de asemenea voi folosii un mutex. Toate acestea se intampla cum s-a acceptat o noua conexiune:

server.cpp

```
if (select(nfds + 1, &readfds, NULL, NULL, &tv))
        {
            len = sizeof(from);
            bzero(&from, sizeof(from));
            client = accept(sd, (struct sockaddr *)&from, &len);
            if (client < 0)
            {
                perror("Eroare la accept()");
            }
            FD_SET(client, &actfds);
            {
                lock_guard<mutex> lock(mutx);
                users_connected.push_back(client);
            }
            if (nfds < client)
                nfds = client;
            pthread_t thread_user;
            thData *td = new thData();
            td->users_connected = &users_connected;
            td->mutx = &mutx;
            td->cl = client;

            if (pthread_create(&thread_user, NULL, handleUser, td) ==
                -1)
            {
                perror("Eroare la creare thread!");
            }
            pthread_detach(thread_user);
        }
```

De asemenea cum am mentionat mai devreme voi avea un timer care verifica numarul de utilizatori conectati simultan la server.

client.cpp

```
QObject::connect(timer_users, &QTimer::timeout, [&]()
        {
              char users_command[100] = "numberusers";
              char number_of_users[10];
              if (write(sd, users_command, strlen(users_command)) == -1)
                        {
                            printf("Eroare la write()!");
                        }
               bzero(number_of_users, sizeof(number_of_users));
               if (read(sd, number_of_users, sizeof(number_of_users)) ==
                   -1)
                        {
                            printf("Eroare la read()!");
                        }
                number_of_users[strlen(number_of_users)] = '\0';
                int nr_users = atoi(number_of_users);
                printf("%d\n", nr_users);
                active_usersTable->setRowCount(0);
       for (int i = 0; i < nr_users; i++)
               {
                    string num = to_string(i + 1);
                    active_usersTable->insertRow(i);
                    string name = "User";
                    name += num;
                    QString username = QString::fromStdString(name);
                    active_usersTable->setItem(i, 0, new QTableWidgetItem(
                        username));
                } });
```

server.cpp

```
bzero(buffer, sizeof(buffer));
if(read(fd, buffer, sizeof(buffer)) < 0) {
    close(fd);
    FD_CLR(fd, &actfds);
    users_connected.erase(remove(users_connected.begin(),
       users_connected.end(), fd), users_connected.end());
    perror("Eroare la read()\n");
}
bzero(message_to_send, sizeof(message_to_send));
if(strcmp(buffer, "numberusers") == 0) {
    int counter_users = users_connected.size();
    string temp_num = to_string(counter_users);
    if(write(fd, temp_num.c_str(), temp_num.size() + 1)) {
        printf("Eroare la write() la nr useri");
    }
}
```

Pentru a putea realiza eficient comunicarea intre utilizatorii conectati asa cum am precizat mai sus, stochez mesajele trimise si utilizatorii care le trimit intr-un fisier de tip *JSON*. Mai jos o sa pun bucatile cele mai importante de cod din tot acest proces:

Thread-ul care se ocupa de manipularea datelor:

```
void *saveInChatLog(void *arg)
{
    thData *td = (thData *)arg;
    vector<pair<string, string>> fullChatLog = openChatLogJSON();

    pair<string, string> temp_pair;
    temp_pair.first = td->username;
    temp_pair.second = td->message;

    fullChatLog.push_back(temp_pair);

    addMessageToJSON(fullChatLog);
    return NULL;
}
```

Observam si doua functii, una care transmite chat-ul actual sub forma de vector de perechi (o pereche este formata din numele utilizatorului si mesajul transmis de acesta), iar cealalta functie adauga un nou mesaj in fisierul ce salveaza "chat log-ul".

Functia openChatLogJSON()

```
vector<pair<string, string>> openChatLogJSON()
{
    vector<pair<string, string>> temp_list;

    ifstream file("/home/razvan/Desktop/Proiect_RC/chat_log.json");

    if (!file.is_open())
    {
        perror("Eroare la deschiderea users_on_file.json");
    }

    ostringstream buffer;
    buffer << file.rdbuf();

    string json_users_content = buffer.str();

    size_t start = 0, end = 0;

    while ((start = json_users_content.find("{", end)) != string::npos)
    {
        pair<string, string> user_and_message;

        start = json_users_content.find("\"username\":", start);
        if (start == string::npos)
```

```
            break;

        start = json_users_content.find("\"", start + 10) + 1;
        end = json_users_content.find("\"", start);
        user_and_message.first = json_users_content.substr(start, end -
            start);

        start = json_users_content.find("\"message\":", start);
        if (start == string::npos)
            break;

        start = json_users_content.find("\"", start + 9) + 1;
        end = json_users_content.find("\"", start);
        user_and_message.second = json_users_content.substr(start, end
           - start);

        temp_list.push_back(user_and_message);
        end = json_users_content.find("}", end) + 1;
    }
    return temp_list;
}
```

Functia addMessageToJSON()

```
void addMessageToJSON(vector<pair<string, string>> temp_vector)
{
    ofstream file("/home/razvan/Desktop/Proiect_RC/chat_log.json");
    if (!file.is_open())
    {
        perror("Eroare␣la␣deschiderea␣fisierului␣chat_log.json");
    }

    file << "[\n";
    bool first = true;
    for (auto i : temp_vector)
    {
        if (!first)
        {
            file << ",\n";
        }
        first = false;

        file << "␣␣{\n";
        file << "␣␣␣\"username\":␣\"" << i.first << "\",\n";
        file << "␣␣␣\"message\":␣\"" << i.second << "\"\n";
        file << "␣␣}";
    }

    file << "\n]\n";
    file.close();
}
```

In final, cea mai importanta functionalitate a proiectului din punctul meu de vedere si cea pe care se bazeaza doua din cele trei functionalitati ale proiectului se refera la transferul binar al unui fisier, reusind astfel sa putem transmite orice fel de fisier. Mai jos o sa arat cum transmit binar un fisier, impartindu-l in bucati. *bucata de cod este extrasa din server.cpp*

```
FILE *file = fopen(path_to_save.c_str(), "wb");
                if (!file)
                {
                    perror("Eroare la deschiderea fisierului!");
                }

                size_t file_size;
                if (select(td->cl + 1, &readfds, nullptr, nullptr, &tv)
                   )
                {
                    if (read(td->cl, &file_size, sizeof(file_size)) ==
                       -1)
                    {
                        perror("Eroare la read()");
                    }

                    char buffer_file[1024];
                    size_t current_bytes_received = 0;

                    while (current_bytes_received < file_size)
                    {
                        size_t bytes_read = read(td->cl, &buffer_file,
                           sizeof(buffer_file));
                        if (bytes_read == -1)
                        {
                            perror("Eroare la read()");
                        }
                        fwrite(buffer_file, 1, bytes_read, file);
                        current_bytes_received += bytes_read;
                        cout << current_bytes_received << '/' <<
                           file_size << '\n';
                    }
                }
                fclose(file);
```

5. Concluzii

Consider ca acest proiect poate reprezenta o unealta esentiala in realizarea comunicarii dintre studentii in intermediul virtual, avand ca functionalitati: posibilitatea de a trimite mesaje text in timp real, verificarea daca alti utilizatori sunt conectati la server, stocarea si partajarea de fisiere si modificarea si sincronizarea in timp real a fisierelor.

Insa exista anumite aspecte ce ar putea fi imbunatatite sau adaugate:

1) Crearea mai multor "camere" pentru diverse proiect, dar acelasi cont.

2) Folosirea unei baze de date mai dezvoltate pentru stocarea anumitor date

3) Sincronizarea in "Live Cooperation" sa fie realizata automat

4) Comunicare audio-video

5) Criptarea istoricului mesajelor si a conturilor utilizatorului

6. Referinte bibliografice

- Informatii despre utilizarea bibliotecii grafice: `https://wiki.qt.io/Main`

- Exemple de utilizare server TCP concurent: `https://edu.info.uaic.ro/computer-networks/files/NetEx/S9/servTcpCSel.c`

- Articol despre fisiere binare: `https://www.cprogramming.com/tutorial/cfileio.html`

- Exemple de utilizare a thread-urilor: `https://edu.info.uaic.ro/computer-networks/files/NetEx/S9/servTcpCSel.c`

- Compararea a doua fisiere in mod binar: `https://cplusplus.com/forum/general/94032/`