Tema 2 - Rețele de calculatoare

Chipăruș Alexandru-Denis, Grupa A4

January 17, 2019

1 Introducere

Acest raport prezintă modul în care am realizat proiectul CollaborativeNotepad (A). Am ales acest proiect deoarece mi s-a părut foarte interesant, deși este destul de complex la capitolul implementare.

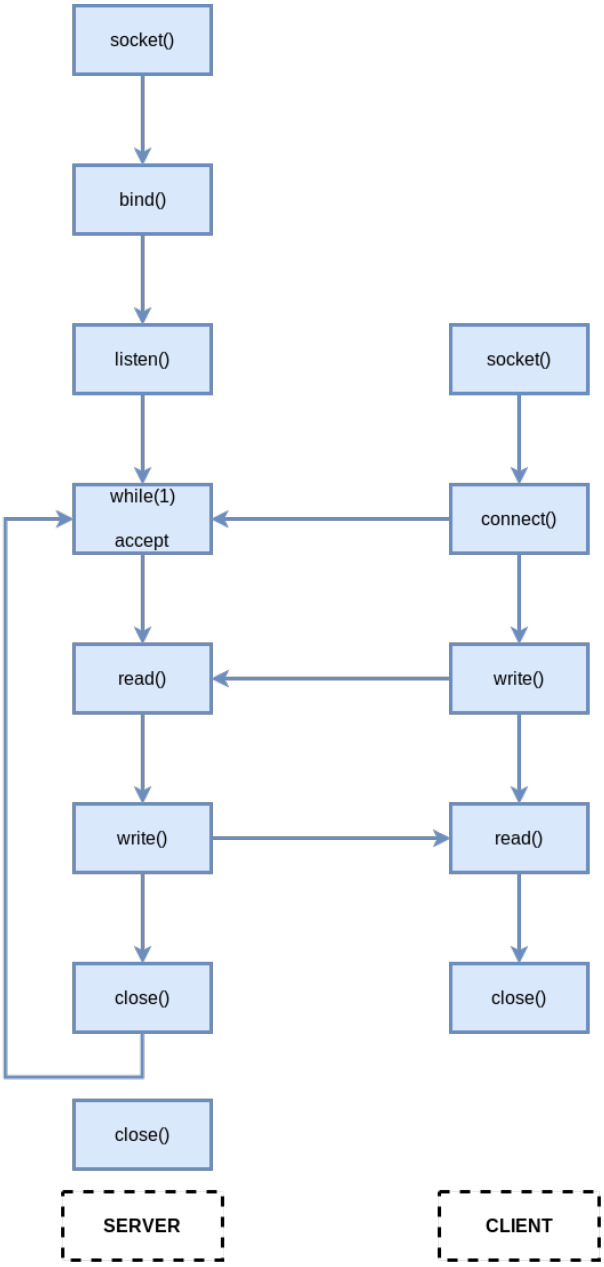
În continuare voi prezenta tehnologiile utilizate, arhitectura aplicației (așa cum am conceput-o în urma documentării) și detaliile de implementare (codul implementat până în acest moment) specifice aplicației.

2 Tehnologiile utilizate

Pentru această aplicație am ales să folosesc modelul client-server TCP concurent și am implementat un server de tip ,,multi-threading". Cu alte cuvinte, serverul acceptă clienți și creează câte un thread pentru fiecare client acceptat care se ocupă de servirea acelui clienți.

Motivul pentru care am ales acest tip de implementare a serverului a fost faptul că aplicația finală trebuie să poată servi mai mulți clienți simultan (mai mult, pot exista mai multe sesiuni de lucru pe fișiere diferite în același timp, deci, un server iterativ nu ar fi fost o soluție fiabilă deoarece nu ar permite editarea ,,simultană" a fișierelor text) și de asemenea protocolul TCP asigură transmiterea corectă a datelor (fără erori, iar datele sunt primite în ordniea în care au fost trimise), evitând astfel coruperea fișierelor ce sunt editate de către clienți.

De asemenea, pentru a avea o interfață grafică acceptabilă pentru ca orice client să poată edita textul cu ușurință, am folosit QT framework (împreună cu QT Creator). Țin să precizez că toate funcțiile de comunicație le-am implementat utilizând metode predate la curs (și nu clase specializate în acest sens din QT framework). Input-ul clientului va fi afișat pe ecran dar în același timp trimis și la server pentru a face modificarea în fișierul text pe care îl editează. Astfel se va simula acțiunea de editare a unui fișier.



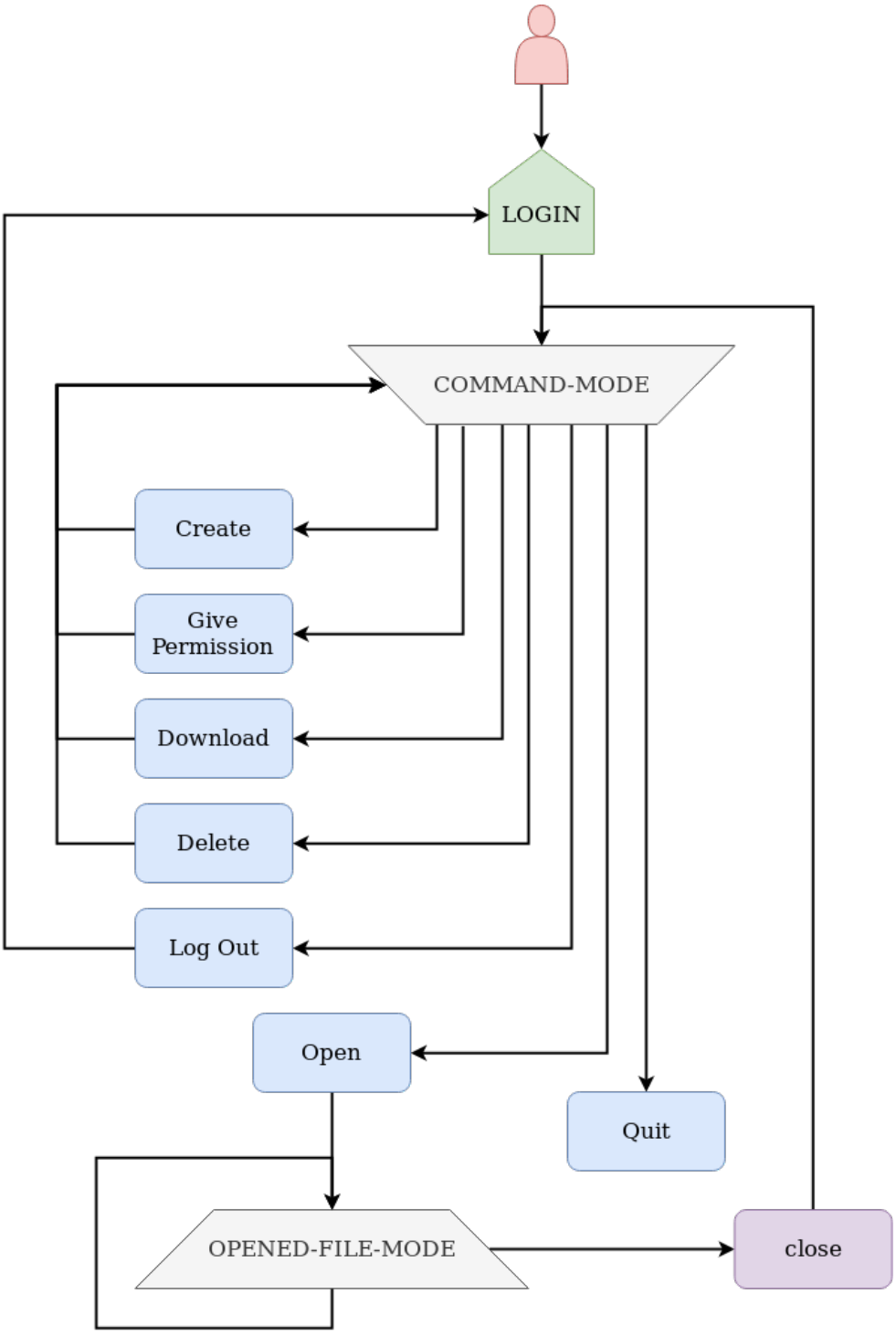
MODELUL CLIENT/SERVER TCP

3 Arhitectura aplicatiei

Aplicația va avea o structură clasică, în care server-ul va oferi clienților diverse servicii, cum ar fi: creare de fișiere (în spațiul de memorie al server-ului), ștergere de fișiere, download de fișiere, editare de fișiere (care se poate face simultan cu un alt client, daca acestui client i s-au oferit drepturile de a accesa fișierul respectiv de către creatorul fișierului, care are grad de administrator). De asemenea, clienții nu vor putea utiliza aplicația decât dacă se autentifică mai întâi.

Pentru client, există două stări în care se poate afla la un moment dat: ,,command-mode” (când execută comenzi; este prima stare în care se află înainte și după autentificare) și ,,opened-file-mode” (în care intră utilizând comanda ,,open” și în care are posibilitatea de a edita fișierul).

Mai jos am reprezentat printr-o diagramă modul de utilizare al aplicației :



Pentru a rezolva eventuale conflicte de editare simultană și corupere a fișierului pe care lucrează doi clienți, fiecare client va stoca într-o variabilă ($tooFastCounter$) numărul de operații realizat până la următorul update de la server. Dacă următorul update de la server este de tip ,,peace", $tooFastCounter$ este setat la 0 și nu se face nimic în plus pe lângă aplicarea operației din update-ul de la server. Dacă update-ul este de tip ,,conflict", atunci serverul a interceptat în același timp două operații de la cei doi clienți și a trimis clientului a cărui operație a fost amânată cele două operații (mai întâi operația de la celălalt și apoi operația sa) și celuilalt, doar operția trimisă de cel amânat. Clientul ce primește două operații realizează undo de câte ori este necesar (în cazul în care a mai trimis între timp și alte operații) și aplică cele două operații primite de la server în ordinea primită (iar acele operații pe care eventual le-a trimis până să primească ,,conflict", le va primi în viitor de la server).

Pentru a stoca informațiile am utilizat fișiere de tip JSON, atat pentru sotcarea utilizatorilor cât și pentru stocarea numelor fișierelor înregistrate în aplicație.

```
int main()
{
    MAX_ACTIVE_USERS = 10000;
    MAX_ACTIVE_SESSIONS = 5000;


    struct sockaddr_in serverAddr;
    struct sockaddr_in clientAddr;

    int sockDescriptor;
    if ((sockDescriptor = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            perror("[SERVER] | [ERROR] : socket()\n");
            return errno;
        }

    int optval = 1;
    setsockopt(sockDescriptor, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));

    bzero(&serverAddr, sizeof(serverAddr));
    bzero(&clientAddr, sizeof(clientAddr));

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sockDescriptor, (struct sockaddr *)&serverAddr, (socklen_t)sizeof(struct sockaddr)) == -1)
        {
            perror("[SERVER] | [ERROR] : bind()\n");
            return errno;
        }

    if (listen(sockDescriptor, LISTEN_QUEUE_SIZE) == -1)
        {
            perror("[SERVER] | [ERROR] : listen()\n");
            return errno;
        }


    ClientsManager clientM(PORT, sockDescriptor, clientAddr, MAX_ACTIVE_USERS);
    clientM.startCM();


    return 0;
}
```

Funcția main din server creează un ClientsManager care acceptă clienți

```
static void* serveClient(void* arg)
{
    int clientDescriptor = *((int*)arg);
    pthread_detach(pthread_self());

    CommandMode cmdM(clientDescriptor);
    cmdM.activateCMMode();

}


class ClientsManager
{

private:

    int PORT;
    int sockDescriptor;
    struct sockaddr_in clientAddr;


public:

    int maxActiveUsers;
    int maxActiveSessions;


    ClientsManager() = delete;
    ClientsManager(int PORT, int sockDescriptor, struct sockaddr_in clientAddr, int maxActiveUsers);
    ~ClientsManager();

    void startCM();


};
```

Clasa ClientManager și funcția rulată de threadul fiecărui client

```
map<int, pthread_t> threadIds;


int idThread = 0;
while (1)
    {
        cerr << "[SERVER] SERVER IS RUNNING (PORT:" << PORT << ")..." << endl;
        fflush(stdout);

        int clientDescriptor;
        int clientAddrLen = sizeof(clientAddr);

        clientDescriptor = accept(sockDescriptor, (struct sockaddr *)&clientAddr, (socklen_t *)&(clientAddrLen));

        if (clientDescriptor < 0)
            {
                perror("[SERVER] | [ERROR] : accept()\n");
                continue;
            }

        cerr << "[SERVER] | [CONNECTED] : CLIENT " << endl;
        fflush(stdout);

        threadIds[idThread] = pthread_t();

        int error;
        error = pthread_create(&(threadIds[idThread]), nullptr, &serveClient, &clientDescriptor);
        if (error != 0)
            {
                perror("[SERVER] | [ERROR] : THREAD CAN'T BE CREATED :\n");
                perror(strerror(error));
            }

        ++idThread;
    }
```

ClientsManager va accepta mai apoi fiecare client și va crea un thread pentru acesta

```
string SERVER_IP = "127.0.0.1";
unsigned int PORT = 2024;

int main(int argc, char *argv[])
{
    int sockDescriptor;
    struct sockaddr_in serverAddr;

    if ((sockDescriptor = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            perror("[CLIENT] | [ERROR] : socket()\n");
            return errno;
        }

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = inet_addr(SERVER_IP.c_str());
    serverAddr.sin_port = htons(PORT);                                    ⚠ implicit conversio

    if (connect(sockDescriptor, (struct sockaddr *)&serverAddr, (socklen_t)sizeof(struct sockaddr)) == -1)
        {
            perror("[CLIENT] | [ERROR] : connect()\n");
            return errno;
        }

    string messageToServer;



    QApplication a(argc, argv);

    LoginWindow loginWindow(nullptr, sockDescriptor);
    CommandModeWindow cmdWindow(nullptr, sockDescriptor);
    EditMode editWindow(nullptr, sockDescriptor);


    loginWindow.setCmdWindow(&cmdWindow);
    cmdWindow.setLoginWindow(&loginWindow);
    cmdWindow.setEditWindow(&editWindow);
    editWindow.setCmdWindow(&cmdWindow);

    loginWindow.show();




    return a.exec();

}
```

Funcția main din client care se connectează la server și apoi creează cele 3 ferestre (LoginWindow, CommandModeWindow și EditWindow)

```
class LogInOutManager
{

private:
    fstream usersFile;

    Communication commObj;
    string username;
    string password;

    int activeUserID;

    bool checkExistance();

    void addUser();

public:
    LogInOutManager() = delete;
    LogInOutManager(string username, string password, int sockDescriptor);

    bool LogIn();

    bool LogOut();

    bool Register();

    string getUserName();
    int getActiveUsrID();



};
```

LogInOutManager din server care se ocupă de login și logout

```
#ifndef LOGINWINDOW_H
#define LOGINWINDOW_H

#include <QMainWindow>
#include <QShowEvent>
#include <QCloseEvent>

#include "communication.h"
#include "closedialog.h"


namespace Ui
{
class LoginWindow;
}

class LoginWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit LoginWindow(QWidget *parent = nullptr, int serverDescriptor = 0);
    ~LoginWindow();


    void setCmdWindow(QMainWindow * cmdWindow);

private:
    Ui::LoginWindow *ui;

    QMainWindow *cmdW = nullptr;

    int serverDescriptor;

    bool loggedIn = false;

    bool checkLogin();
    bool checkRegister();

private slots:

    void showEvent(QShowEvent *event);
    void closeEvent (QCloseEvent *event);

    void on_loginButton_clicked();

    void on_registerButton_clicked();


};

#endif // LOGINWINDOW_H
```

LoginWindow din client care se ocupă de login și înregistrare

```
class CommandMode
{

private:

    int clientDescriptor;

    int activeUserID = -1;

    Communication commObj;

    bool loggedIn = false;
    string username;
    string fileName;

    FileManager fm;
    LogInOutManager log = {"usr", "pass", clientDescriptor};

    void processCommand(string messageFromClient);
    AvailableCommands convertToEnum(string message);

public:

    CommandMode() = delete;
    CommandMode(int clientDescriptor);

    void activateCMMode();

};
```

CommandMode din server ce se ocupă de procesarea comenzilor primite de la client

```
class CommandModeWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit CommandModeWindow(QWidget *parent = nullptr, int serverDescriptor = 0);
    ~CommandModeWindow();

    void setDescriptor(int descriptor);
    void setLoginWindow(QMainWindow* loginWindow);
    void setEditWindow(QMainWindow* editWindow);

    void updateFileExplorer(deque<QString> & files);

private:
    Ui::CommandModeWindow *ui;

    QMainWindow * loginW = nullptr;
    QMainWindow * editW = nullptr;

    int serverDescriptor;

    bool logout = false;

    NewFileDialog nfd;

private slots:

    void showEvent(QShowEvent * event);
    void closeEvent (QCloseEvent *event);

    void on_openButton_clicked();

    void on_createButton_clicked();

    void on_downloadButton_clicked();

    void on_deleteButton_clicked();

    void on_quitButton_clicked();

    void on_logoutButton_clicked();

    void on_permButton_clicked();

};
```

CommandModeWindow din client care se ocupă de gestionarea ferestrei CommandMode cu toate comenzile pentru client

```
struct editModeThArgs
{
    int clientId;
    int clientDescriptor;
    ActiveSession * currentSession;
};



class EditMode
{

private:
    int clientDescriptor;
    int clientId;

    ActiveSession * currentSession;


    Communication commObj;

public:

    EditMode();

    void activateEMode();

    void setClientDescriptor(int clientDescriptor);
    void setClientId(int clientId);
    void setSession(ActiveSession * currentSessionParam);

    static void * sendUpdateToClient(void * arg);
    pthread_t tid;


};
```

EditMode ce se ocupă de coordonarea dintre SessionManager și client în timpul editării

```
class EditMode : public QMainWindow
{
    Q_OBJECT

public:
    pthread_mutex_t lock;

    explicit EditMode(QWidget *parent = nullptr, int serverDescriptor = 0);
    ~EditMode();
    void setDescriptor(int serverDescriptorParam);

    Operation* getOperation();
    void setOperation(Operation &opParam);
    void setCmdWindow(QMainWindow * cmdWindow);

private slots:
    void showEvent(QShowEvent *event);
    void closeEvent (QCloseEvent *event);

    void textFieldChanged(int position, int removed, int inserted);

    void updateProcessing();

private:
    Ui::EditMode *ui;
    QTimer timer;

    Operation op;
    QMainWindow * cmdW = nullptr;



    void processServerResponse(string fromServer);
    void applyOperation(Operation *op);


    static void* createThread(void *args);
    void* update(void);

    pthread_t tid;

};
```

EditMode din client care se ocupă de coordonarea textului scris de client și textul de la server

```
class ActiveSession
{
    pthread_mutex_t lock;

    string user[2];
    string fileName;

    Operation fromUser[2];
    bool fromUserChanged[2];

    deque<Operation> toUser0;
    deque<Operation> toUser1;

    bool toUserChanged[2];

public:

    ActiveSession();
    ~ActiveSession();

    void setFileName(string fileName);
    bool isFull();

    int addUser(string username);
    void removeUser(int clientId);

    bool fromClientChanged(int idClient);
    bool toClientChanged(int idClient);

    void setOperationFromClient(int idClient, Operation &op);
    void setOperationToClient(int idClient, deque<Operation>& ops);

    Operation& getOperationFromClient(int idClient);
    deque<Operation>& getOperationToClient(int idClient);

    void lockFunction();
    void unlockFunction();

    bool isUser0();
    bool isUser1();

    void setClear();
    bool isClear();

    string getUser(int userId);

    friend class SessionManager;
};
```

Clasa ce reprezintă o sesiune de lucru pe care se va opera din SessionManager

SessionManager ce operează clasa ActiveSession și realizează și scrierile în fișierele de pe disc

```
#ifndef OPERATION_H
#define OPERATION_H

#include <iostream>
#include <string>


using namespace std;

enum TypeOfOp
{
    EXIT = 0,
    DEFAULT = 1,
    INSERT = 2,
    REMOVE = 3,
    MIXED = 4
};


class Operation
{
private:
    TypeOfOp type;
    int noChars;
    int position;
    string charsInserted;

public:

    Operation();
    Operation(TypeOfOp type, int noChars, int position, string &chars);

    void setType(TypeOfOp operationType);
    void setNoChars(int operationNoChars);
    void setPosition(int operationPosition);
    void setCharsInserted(string inserted);

    TypeOfOp getType();
    int getNoChars();
    int getPosition();

    string getCharsInserted();
};

#endif // OPERATION_H
```

Clasa ce reprezintă operațiile ce se pot face asupra textului

```
class FileManager
{
private:
    fstream docsFile;
    fstream permissionsFile;
    fstream usersFileStream;

    json filesJSON;
    json currentFileJSON;
    json::iterator currIterator;

    string emptyStr = "";

    string fileName;
    deque<string> admins;
    deque<string> permitted;
    string fileContent;

    void createFileOnDisk();
    void deleteFileFromDisk();

    deque<string> userFiles;

public:
    FileManager();

    bool fileExists(string fileNameParam);
    bool findFile(string fileNameParam);
    void clearInfos();

    bool createFile(string fileNameParam, string creatorUsername);
    bool deleteFile(string fileNameParam);
    string & downlodFile(string fileNameParam);


    bool addPermission(string filename, string userToAdd);

    string getFileName();
    deque<string>& getAdmins();
    deque<string>& getPermitted();

    deque<string>& getAllUserFiles(string username);


};
```

Clasa FileManager care se ocupă de adăugare și ștergere de fișiere de pe disc

```
class Communication
{
private:

    int commDescriptor;


public:

    Communication();
    Communication(int descriptor);

    void sendStringToDescriptor(string message);
    string recvStringFromDescriptor();

    void sendOperationToDescriptor(Operation &operation);
    Operation* recvOperationFromDescriptor();

    bool isMessageReceived();
    void sendByte(char byteValue);
    void setDescriptor(int descriptor);

};
```

Clasa Communication care se ocupă de toate operațiile de comunicație dintre server și client

4 Concluzii

După părerea mea, acest proiect este complex dar în același timp foarte atractiv pentru toți cei care vor să învețe cum se realizează un editor de text în C/C++ sau cum se simulează partea de editare simultană.

După modul în care am proiectat aplicația, aceasta ar putea fi îmbunătățită dacă s-ar crește numărul de clienți pe sesiune de lucru, dacă s-ar oferi funcționalități în plus, cum ar fi editare de fișiere cu tabele, text formatat HTML (care se poate realiza cu clasa QTextEdit din QT framework ce acceptă și text formatat HTML) și de asemenea, dacă s-ar realiza comunicația criptată.

Bibliografie

[1] Pagina Cursului,
https://profs.info.uaic.ro/ computernetworks/index.php

[2] Pagina de la laborator,
https://sites.google.com/view/fii-rc

[3] Tutorial creare editor de text în C,
https://viewsourcecode.org/snaptoken/kilo/01.setup.html

[4] Idee de proiect: construirea a unei aplicații de tip ,,collaborative editor",
https://www.geeksforgeeks.org/project-idea-collaborative-editor-framework-real-time/

[5] Documentație librăria ncurses,
http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/

[6] O postare de pe site-ul stackexchange.com,
https://softwareengineering.stackexchange.com/questions/288025/reading-file-during-write-on-linux

[7] File locking, Wikipedia,
https://en.wikipedia.org/wiki/File_locking

[8] Documentație QT framework,
http://doc.qt.io/