International University of Sarajevo

Faculty of Engineering and Natural Sciences

CS304 Project report

Programmable macropad

Students:

Muhammed Mušanović

Harun Tucaković

Mentor:

Prof. Ali Abd Almisreb, PhD

Sarajevo, May 2020

Contents

1. Abstract

Project goal was to create a programmable macropad which is going to provide user with additional functionalities that are not available on standard keyboards. Meaning that, we are providing our users with set of additional keys/buttons that can be completely customizable and programmable using simple GUI (Graphical User Interface) that will come with a driver for the macropad.

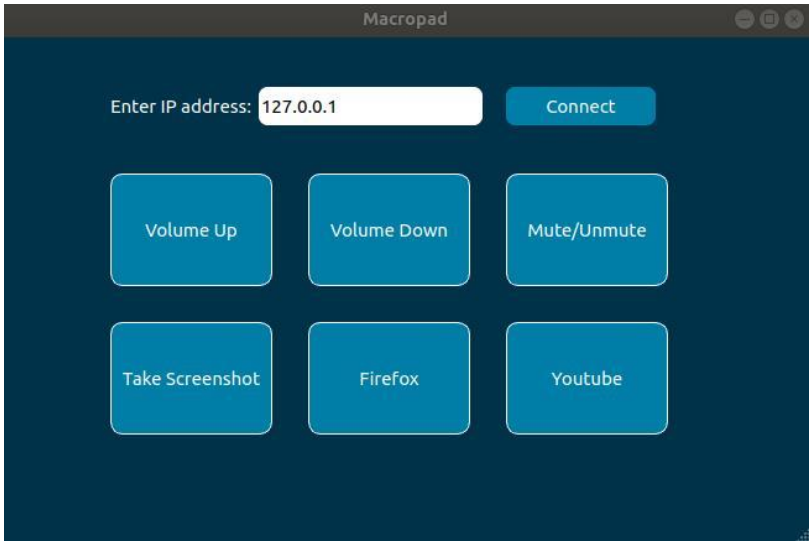
NOTE: Due to coronavirus outbreak and quarantine that closed the whole world during the semester we were unable to get our hands on the hardware parts needed for creating actual physical macropad. Therefore, we created GUI simulation of the macropad. It is a software that acts like physical macropad would behave and provides a possibility to fully test the application and project as a whole.

2. Introduction

This project consists out of two applications: Macropad (simulation of the physical macropad) and Macropad Settings (main macropad driver with GUI for configuring the Macropad). These applications are developed for Unix based operating systems. They are designed to communicate over wireless or ethernet connection on LAN (Local Area Network). They can also be ran and tested on the same machine (use localhost as IP address).

2.1 Macropad

As mentioned before, due to worldwide lockdown during this semester and inability to get all hardware parts needed to create physical Macropad keyboard, the next best thing we could have done is to create software that will completely simulate the Macropad. Hence, we created this Macropad simulation application.



Macropad simulation application consists out of simple GUI and server (for effective communication with Macropad Settings application).

When Macropad application is launched, first thing that needs to be done is to enter IP address of the machine which is running Macropad Settings application and connect to the server that is integrated into Macropad Settings. After connecting to the server, Macropad application is ready to use.

Take note that this button configuration seen on the image above is „Default“ button configuration that can be completely customized (not visually, but functionally). After connection with Macropad Server is established, user preset button configuration will be loaded. Details about inner workings of this application will be further discussed in following sections.

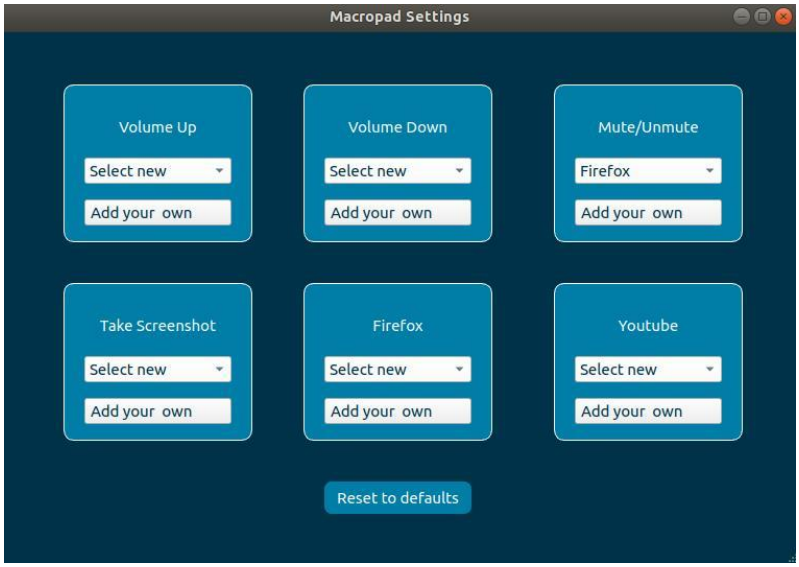
2.2 Macropad Settings

Macropad Settings application also consists out of GUI and server part. But, this app acts like driver for Macropad and provides much more functionalities. On the launch the Macropad Settings application will wait for the connection that needs to be initialized from Macropad. After the connection is established Macropad Settings app will:

- Execute commands binded with buttons on the Macropad. Execution of these commands is done in shell by default on most Linux systems. To specify specific shell that one wants to execute

command in, simply follow this template: <shell_name> -c \“<shell_command>\“ (e.g. „ *bash -c \“apt-get upgrade -y\“ “*)

- Provides one with simple GUI to customize names and functionalities of buttons on the macropad. As of now, there are six in-built command/functionalities from which user can choose. In-built options are available in the drop list called „*Select new*“ that can be see on the picture to the right. But there is also a way for user to enter custom command. Pressing the „*Add your own*“ button will open up the dialog to enter custom name and custom command for the button. When we say „*command“,* we mean: command in the shell that your system uses as default (We explained previously how to enter shell specific commands). One should be carefull with distribution and desktop environment compatibility of commands that one enters. One should also try not to run the application as root if not completely aware of risks. „*Reset to defaults*“ button will clear saved user preset and reset all button names and functionalities to default preset shown on the pictures above.



That would be all about functionalities of these two applications. In the next section we will discuss details about their implementation and inner workings.

3. Method and implementation

Applications that are presented in this report are completely developed in C++ programming language. Both graphical user interfaces are developed using free and open-source C++ toolkit for developing graphical user interfaces called Qt. All internet communication between two applications is programmed in C++ (socket network programming). To be able to continuously run communication server and actual GUI application in parallel as a same program we used multithreading techniques and *pthread* library in C++.

3.1 Macropad

3.1.1 Graphical user interface

Graphical user interface for Macropad application consists only out of main window. On the main window we have one *"QLineEdit"* element which provides user with single line text input (for IP address), and six *"QPushButton"* elements that represent buttons.

Class that represents the main window is called *"StreamDeck"*. As class members we can see: *"ui"* which is main container in the main window, *"buttonNames"* which is vector that holds QString representation of all names of all buttons, *"serverHandle"* is pointer to the thread on which server is running, *"_mutexLock"* is mutex that locks the thread if needed, *"buttonsVector"* is the vector that holds pointer to the actual *QPushButton* objects.

```
class StreamDeck : public QMainWindow
{
    Q_OBJECT

public:
    StreamDeck(pthread_t*, pthread_mutex_t*, QWidget *parent = nullptr);
    void setButtonNames(std::vector<QString>);
    void getNamesFromServer();
    ~StreamDeck();
protected:
    void closeEvent(QCloseEvent *event) override;
private:
    Ui::StreamDeck *ui;
    std::vector<QString> buttonNames;
    pthread_t * serverTHandle;
    pthread_mutex_t * _mutexLock;
    std::vector<QPushButton *> buttonsVector;


private slots:
    void buttonPressed();
    void connectToServer();
};
```

Class methods are:

- *StreamDeck* – constructor that initializes the application main window and class members
- *~StreamDeck* – deconstructor that frees dynamically allocated memory
- *setButtonNames* – method that sets class member *"buttonNames"*
- *getNamesFromServer* – method that contacts Macropad Settings applications and gets list of button names from database
- *closeEvent* – method that is overridden to cancel and join server thread with the main thread when application is about to be closed
- *buttonPressed* – method that is called when one of six buttons are pressed. It checks which button was pressed and sends message to the Macropad Settings application with information which button was pressed
- *connectToServer* – method that is called when *"Connect"* button is pressed. This method just connects to the server at Macropad Settings application and delivers Macropad machine IP address as data

3.1.2 Network communication

There are two types of network communication happening in this application. First and more complex type of network communication is where Macropad application is running server and waiting for messages/updates from Macropad Settings application.

In this case, we are using *"server"* function from *serverConnection.h* file, which is responsible for setting up a server to listen on the port for incoming connections. Process of setting up a server to listen on the specific port is next: first we initialize network socket; we initialize *sockaddr_in* structure to hold information about the server (socket family, port, addesses); then we bind that structure with our socket; set socket to listening for connections state and finally tell the server to store new connections into new socket so we can access them. After the new connection is established, we read and process data and close the connection, returning to the listening state. We can see server setup process on the picture below.

```
void serverSetup(int* _socket, int* opt, size_t optSize, struct sockaddr_in* address, size_t addressLen) {
    // Creating socket file descriptor
    if (((*_socket) = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        std::cerr << "Socket creation failed!" << std::endl;
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(*_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, opt, optSize)) {
        std::cerr << "setsockopt failed!" << std::endl;
        exit(EXIT_FAILURE);
    }

    address->sin_family = AF_INET;
    address->sin_addr.s_addr = INADDR_ANY;
    address->sin_port = htons( PORT );

    // Forcefully attaching socket to the port 8080
    if (bind(*_socket, (struct sockaddr *)address, addressLen) < 0) {
        std::cerr << "Socket binding failed!" << std::endl;
        exit(EXIT_FAILURE);
    }

    if (listen(*_socket, 4) < 0)
    {
        std::cerr << "Listening failed!" << std::endl;
        exit(EXIT_FAILURE);
    }

    std::cout << "Listening on the port " << PORT << "..." << std::endl;
}
```

For this server to run in parallel with our GUI application, we are calling it on the separate thread in the main function. We are doing that using *"pthread_create"* function. And we are sending as a parameter

```
int main(int argc, char *argv[])
{
    pthread_t serverT;
    pthread_mutex_t threadLock;
    QApplication a(argc, argv);
    StreamDeck w(&serverT, &threadLock);

    pthread_create(&serverT, NULL, &server, (void*) &w);
    if (pthread_mutex_init(&threadLock, NULL) != 0) {
        std::cerr << "\n mutex init has failed\n" << std::endl;
        return 1;
    }

    w.show();
    return a.exec();
}
```

pointer to the object of the main window, so our server has access to the main window methods and state. But also, when we are creating main window object called *"w"* we are passing as parameters references to our server thread and mutex lock, so our main window has control over server thread.

This is the server over which Macropad application is receiving all the data and updates from the Macropad Settings application. Second type of network communication that is happening in this application is where Macropad application must send a message to the Macropad Settings application. This communication is simply done by creating a socket, setting up connection to the Macropad Settings server and sending a message with necessary data, after which connection and our socket are closed.

3.2 Macropad Settings

3.2.1 Graphical user interface

Graphical user interface for Macropad Settings application is quite like the GUI of Macropad application, but this one is more complex and has more features. On the main window we have six *"Frames"* that hold *"QLabel"*, *"QComboBox"* and *"QPushButton"* each. Combo box represents drop down list that will let user to select one of the built-in functions for that button. Button below the drop-down list will, on click, open dialog box that will let user enter custom name and command for that button.

```
class StreamDeckSettings : public QMainWindow
{
    Q_OBJECT

public:
    StreamDeckSettings(pthread_t *, pthread_mutex_t *, QWidget *parent = nullptr);
    ~StreamDeckSettings();
    void readConfig();
    void loadButtonInfo();
    void toggleConfig();
    std::vector<std::string> getPreset();
    std::vector<std::string> getDefaultC();
    void setPreset(std::vector<std::string>);
    std::vector<std::string> getButtonNamesVec();
    std::vector<std::string> getButtonCommandsVec();
    std::vector<std::string> splitStr(std::string str, std::string delimiter);
    void setClientAddresses(std::vector<std::string>);


private slots:
    void editbtnsClicked();
    void droplistChange();

    void on_pushButton_released();

protected:
    void closeEvent(QCloseEvent *event) override;

private:
    Ui::StreamDeckSettings *ui;
    QList <QLabel *> labels;
    pthread_t *_serverThread;
    pthread_mutex_t *_threadLock;
    int preset = 0;
    std::vector<std::string> buttonNamesVec;
    std::vector<std::string> buttonCommandsVec;
    std::vector<std::string> clientAddresses;
    void multithreadClientMessages(char mess[2048], size_t size);

};
```

In this application, class that represents main window is called *"StreamDeckSettings"*. Class members of this class are: *"ui"* which is main container in the main window, *"labels"* which is list containing names of labels in the application, *"_serverThread"* is pointer to the *pthread_t* variable which holds reference to the server thread, *"_threadLock"* which is pointer to the mutex that locks server thread if that is needed, *"preset"* is variable that holds information whether default or user preset for buttons is in use currently, *"buttonNamesVec"* and *"buttonCommandsVec"* are vectors holding information about names and commands for each button, *"clientAddresses"* is vector that holds IP addresses of the machine running the Macropad application and it is populated when Macropad is firstly connected to the Macropad Settings.

Class methods are:

- *StreamDeckSettings* – constructor that initializes main window and class members
- *~StreamDeckSettings* – deconstructor which frees dynamically allocated memory
- *readConfig* – method that reads whether *default* or *user* preset is in use currently, this information is read from the file on system
- *loadButtonInfo* – method which loads information about buttons from the file on the system (*default* and *user* presets are saved in files on the system)
- *toggleConfig* – method that changes preset state, it is changed in the class member variable "preset" and in the file that holds that information
- *getPreset* – reads current preset from the file
- *getDeafultC* – reads default preset regardless of which preset is in use right now
- *setPreset* – writes new preset in file
- *getButtonNamesVec* – return vector of button names; it is used on the server, for server to get access to current button names
- *getButtonCommandsVec* – returns vector of button commands; it is used on the server also, for server to get access to current button commands
- *splitStr* – utility method that splits string on specific delimiter and returns vector of tokens
- *setClientAddresses* – sets class member variable *"clientAddresses"* to the new value
- *editbtnsClicked* – registers click on the *"Add your own"* button and handles opening of new dialog for custom button information
- *droplistChange* – registers change on drop-list and handles changing button name and functionality when user chooses new item from the drop-list
- *on_pushButton_released* – registers and handles when user clicks *"Reset to default"* button
- *multithreadClientMessages* – method that sends multiple messages to the Macropad application in parallel

3.2.2 Network communication

Network communication in Macropad Settings application in its essence is very similar to the network communication in the Macropad application, but here we have much more complex data handling and processing on the server. There are two types of network communication, again. We are going to firstly cover one where Macropad Settings application is sending signals and information to the Macropad application.

This kind of communication between applications is quite simple. There is one function that implements it and that function is called *sendNamesToClient* and is placed in "*server.h"* file. For multithreading purposes this function is wrapped in *sendNamesToClientWrapper* function. Also, it is important to mention that *sendNamesToClient* function takes in three arguments, but because of the syntax for creating multithreaded function we can send only one argument to our function wrapper.

```
// We have to send arguments to the function "sendNamesToClientWrapper" as a void*
// Therefore we pack all the arguments in a structure, and send the pointer to this structure to the mentioned function
typedef struct sendToClientArgs {
    char msg[2048];
    size_t size;
    std::string _addr;
} toClientArgs;
```

Hence, we created structure called *"sendToClientArgs"* which has three fields for three arguments that must be passed to *sendNamesToClient* function. Then, we can easily cast pointer to the structure to the (void *) pointer and pass it that way to the wrapper. We can see that structure on the picture on the previous page.

Another type of network communication in this application is server that is created and ran in parallel with the main GUI application. This server is listening for the connections from the Macropad application. When connection is received server checks which button was pressed and based on that it executes appropriate action. There are six default built-in functionalities that are implemented as of now (there will be more default options in future versions of the application; more on that in following sections). Implemented functionalities right now are:

- Volume Up – increases volume output on the machine by 5%
- Volume Down – decreases volume output on the machine by 5%
- Mute/Unmute – mutes/unmutes the volume
- Take Screenshot – takes screenshot of the display
- Firefox – opens new Firefox window
- YouTube – opens new YouTube tab in the Firefox web browser

Shell commands that execute these actions are:

- amixer -D pulse sset Master 5%+ > /dev/null
- amixer -D pulse sset Master 5%- > /dev/null
- amixer -q -D pulse sset Master toggle
- gnome-screenshot
- firefox
- firefox --new-window https://www.youtube.com

3.2.3 Data persistence

Macropad Settings application needs to persist some data to permanent memory, but amount of data stored that way is not substantial, hence we decided to store that data into plain text files. Three text files are important for the application: *config.txt* which stores type of preset that is used in application (default/user preset), *default.txt* stores the default button names and commands and finally *preset.txt* stores user preset (buttons and commands that user customized). These text files need to be in the directory where application is compiled and ran.

4. Results

This project has slightly changed compared to our project proposal that was provided at the beginning of the semester. Changes that were made are due to worldwide coronavirus pandemic that prevented us from getting necessary hardware. But, on the other hand, it provided us with opportunity to create this simulation that works in the same exact way in which physical macropad would work. The most important thing is that we stuck with C++ and important low-level programming features that make this code easily portable to all most popular microcontrollers like Arduino, ESP microcontrollers and even Raspberry Pi. Most complex features in the project, like network programming and multithreading, are done in C like style to ensure easy portability to hardware.

Compatibility of Macropad Settings application and driver with other operating system like Windows and MacOS is also possible and thought of. For compatibility with Windows only socket initialization and setup is necessary (with other simple tweaks in network programming part) and shell commands that are executed by the driver need to be converted to PowerShell commands. For compatibility with MacOS systems only some driver commands need rework.

4.1 Future

Applications developed in this project are designed with expansion and future improvements in mind. Main things that are planned in future updates are: more default built-in commands, number of buttons in the simulation application customizability, running the driver application as a background process, porting Macropad application to android devices, porting Macropad Settings application and driver to Windows and MacOS systems, and of course porting Macropad code to the hardware to create physical macropad.

Third development phase will implement: encrypted network communication, expansion from LAN to Internet communication, implementation of encrypted data storage on local device.