



Lebanese American University

School of Arts and Sciences

Department of Computer Science and Mathematics

CSC430 (Computer Networks)

Assignment #03 (Client-Server Application)

A Report Done By

Adam Kanj

adam.kanj@lau.edu

Mahmoud Joumaa

mahmoud.joumaa@lau.edu

And presented to Dr. Rasha El Khansa

December 2023

Table of Contents

I.	Introduction.....	3
II.	Application Code Skeleton	3
I.	Client.....	3
II.	Server	3
III.	Running the application	4
III.	Additional Features	5
I.	User Authentication From a Database (Text File)	5
II.	User Broadcast	7
III.	User Deposit and Withdraw Actions	8
IV.	Conclusion	10
V.	References.....	10

I. Introduction

For our final CSC430 assignment, we were tasked to design and implement a client-server application with an added layer of extra features.

We opted to design a simple bank-like application that keeps track of users in a database (in the form of a text file, for the sake of maintained simplicity) and allows different users to sign up, log in, deposit money into their accounts, and withdraw money from their accounts. Once a user sign up, a message is broadcasted to all connected clients celebrating a new customer.

The server is an always-on terminal listening for different client requests. It can handle up to 10 clients simultaneously. The clients are represented via simple Graphical User Interfaces, or GUIs, for short, with input labels and buttons instead of a standard terminal screen.

This report describes the application's structure and architecture, the different implemented features, and cites the different documentation we referred to while working on this assignment.

Note that this report does not show screenshots of the source code, but rather focuses on the resulting interfaces and communication. Kindly refer to the heavily commented source code in the submission zip for details regarding the implementation.

II. Application Code Skeleton

I. Client

Starting with the client code, it is a Python script designed to interact with a server-based banking application using a graphical user interface (GUI) built with Tkinter. It establishes a socket connection to the server via a defined IP address and port number. The GUI includes input fields for username, password, and transaction amounts (deposit/withdraw), that expect as input the username and the amount/message separated by a space. Upon user actions (signup, login, deposit, withdraw), the client sends formatted messages to the server, expecting responses displayed in the GUI's text box.

Additionally, it enables a broadcast feature allowing users to send messages to other connected clients through the server by entering text in the broadcast entry and clicking the "Broadcast" button, facilitating communication between users in the banking system.

II. Server

As for the server, the initializations are in the form of a *users* dictionary and a *clients* set to keep track of all users that have an account for our application and all clients that have connected since the server started its current session.

When the server initializes at local host on port 12348, it first binds this IP address with the port and listens to 10 simultaneous client requests.

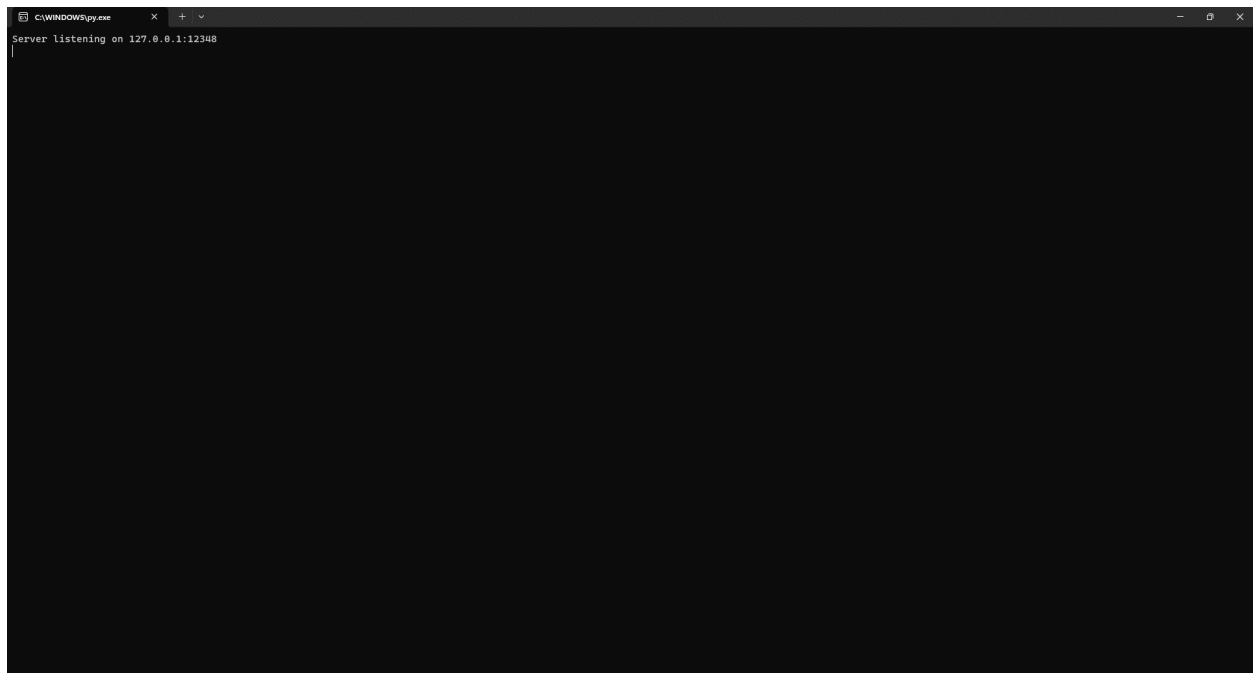
After the server configurations had taken effect, the server reads all users of our banking application from the database (*users.txt* file) and saves them into the working dictionary, keeping track of all current users.

The infinite *while* loop is the part of the code that keeps the server on, ready to accept different client requests. Once the server starts this loop, it keeps track of all clients connecting to it in the *clients* set. This set will later be referenced when broadcasting a message to the different client addresses.

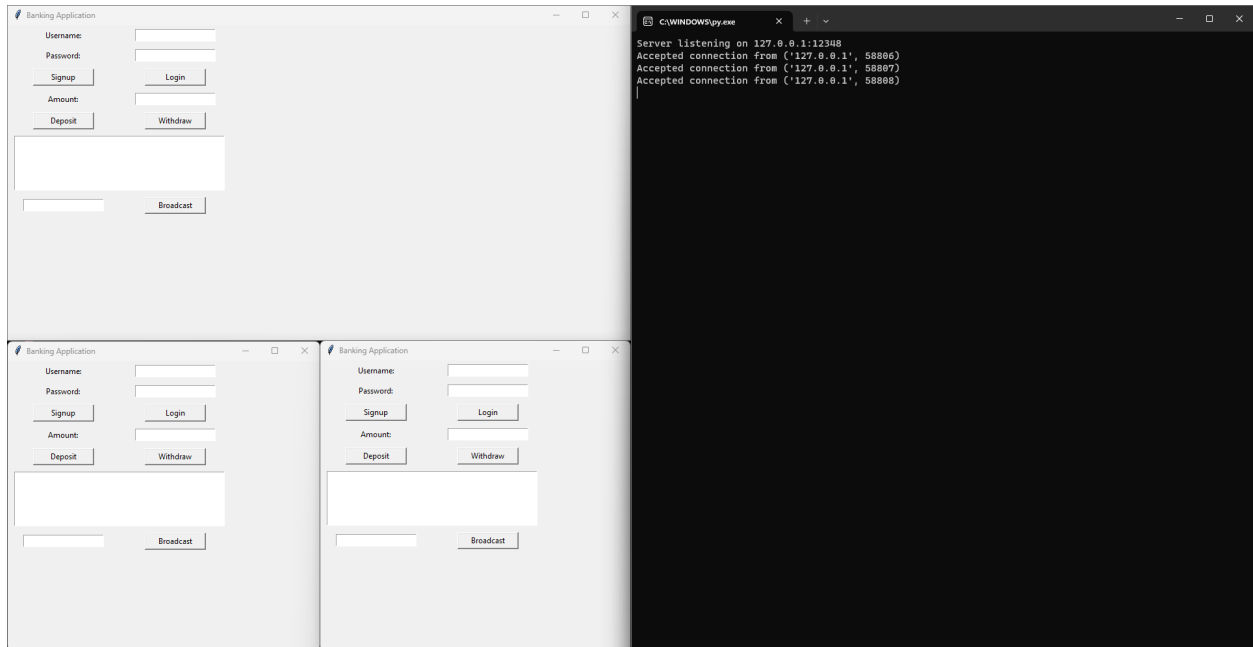
III. Running the application

To run our banking application, we first have to initiate our server component of the application. The server is a simple terminal window. User interaction with this component is fairly limited as it only outputs to the console log messages, so we didn't design or develop a GUI for the server.

Once started, the server should display a message stating that has successfully started and is now listening on the specified configurations.

A screenshot of a terminal window titled 'C:\WINDOWS\system32\cmd.exe'. The terminal output shows 'Server listening on 127.0.0.1:12348' followed by a cursor. The terminal window has a dark background and standard window controls (minimize, maximize, close) in the top right corner.

As for the clients, multiple instances can be run at the same time. Since the user would typically be interacting with this component quite often, we have decided to develop a simple, yet functional, GUI for ease of interaction. Once a client connects to the server, the server displays a message indicating successfully establishing the communication.



The client is in the form of a command terminal running in the background, but the user will be interacting with the GUI. The different parts of the GUI are further described in the following section.

III. Additional Features

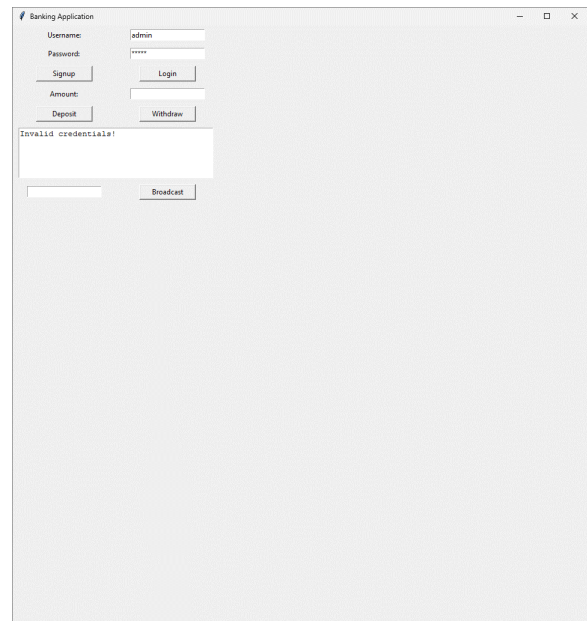
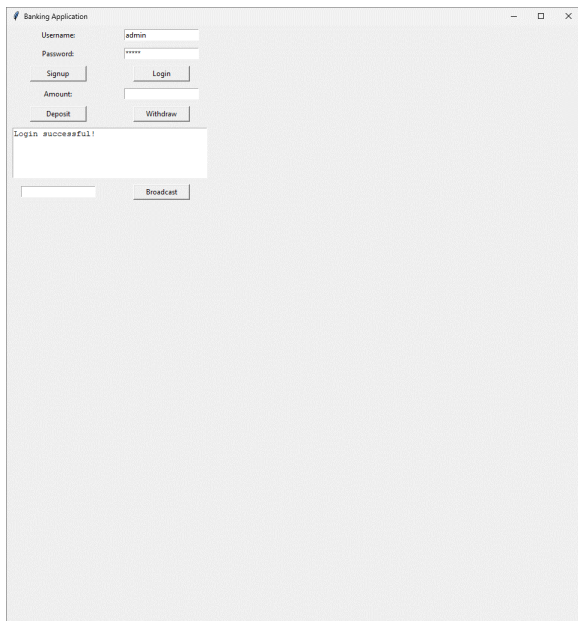
The *handle_client()* function handles the different logic based on the client's request type (if it's a sign up, or log in, withdraw, deposit, or broadcast.)

I. User Authentication From a Database (Text File)

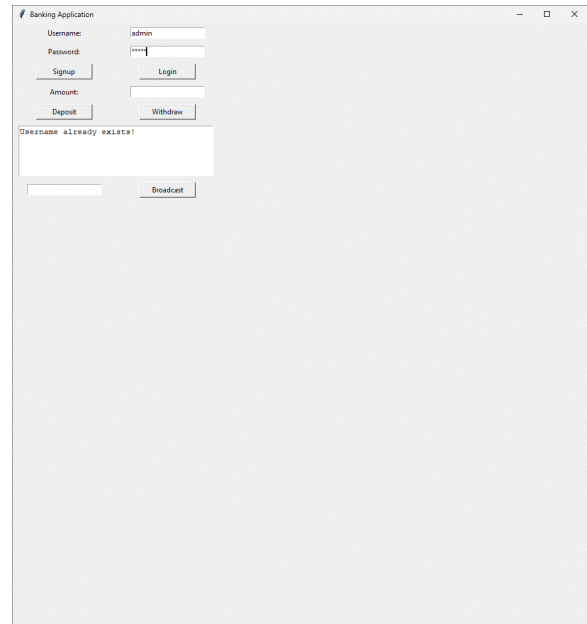
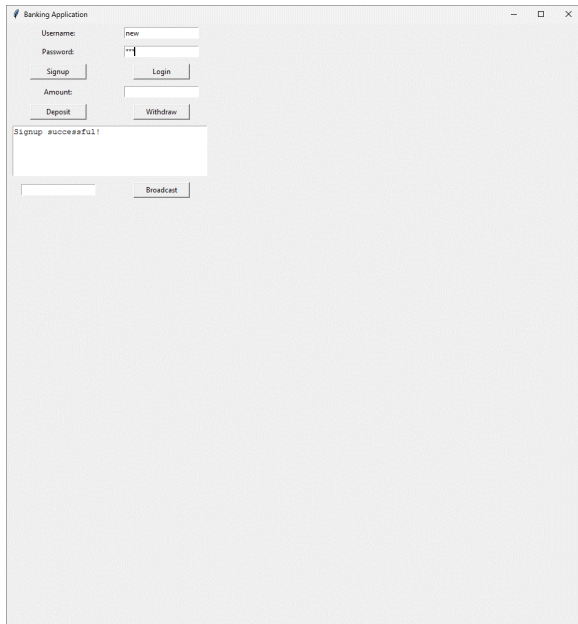
This application's back-end database is represented in the form of a simple text file. Upon startup, the server loads all information from this file. Every line of the file represents a customer in the form of "*username password balance*".



A user may attempt to login using the corresponding credentials. If the credentials don't match or don't exist, an error message is displayed instead of a successful login.

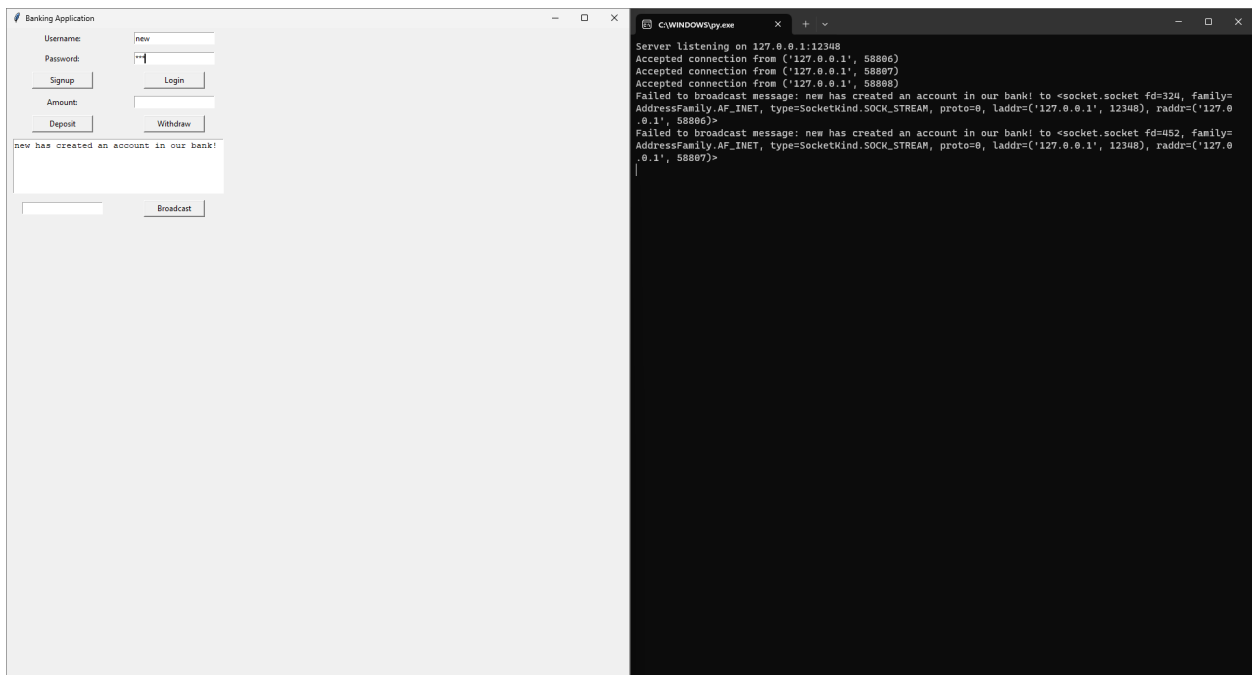


Once a user decides to sign up, if the credentials aren't already used by someone else, the server stores this new information, with a balance of zero, in both the dictionary using the *add_user()* function and then stores it in the file using the *update_users_db()* function.



II. User Broadcast

Whenever a new user signs up and creates an account for our banking application, a message celebrating this new customer is broadcast to all other clients. As some messages might fail, the server displays an error message indicating which clients failed to receive what message. As for the clients, they can use the *broadcast* button to display all broadcasted messages to their address.

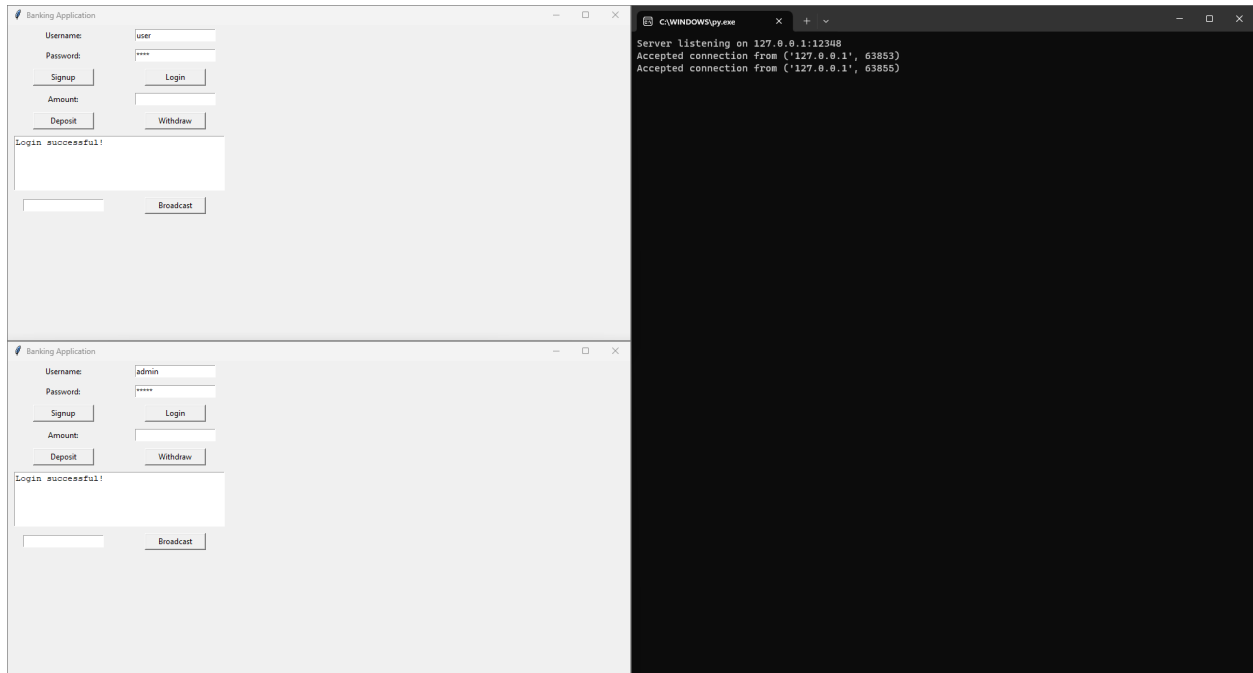


Note that the server displays two error messages as we'd closed the other two clients that were running.

This feature can, of course, be extended to broadcast any message to all clients.

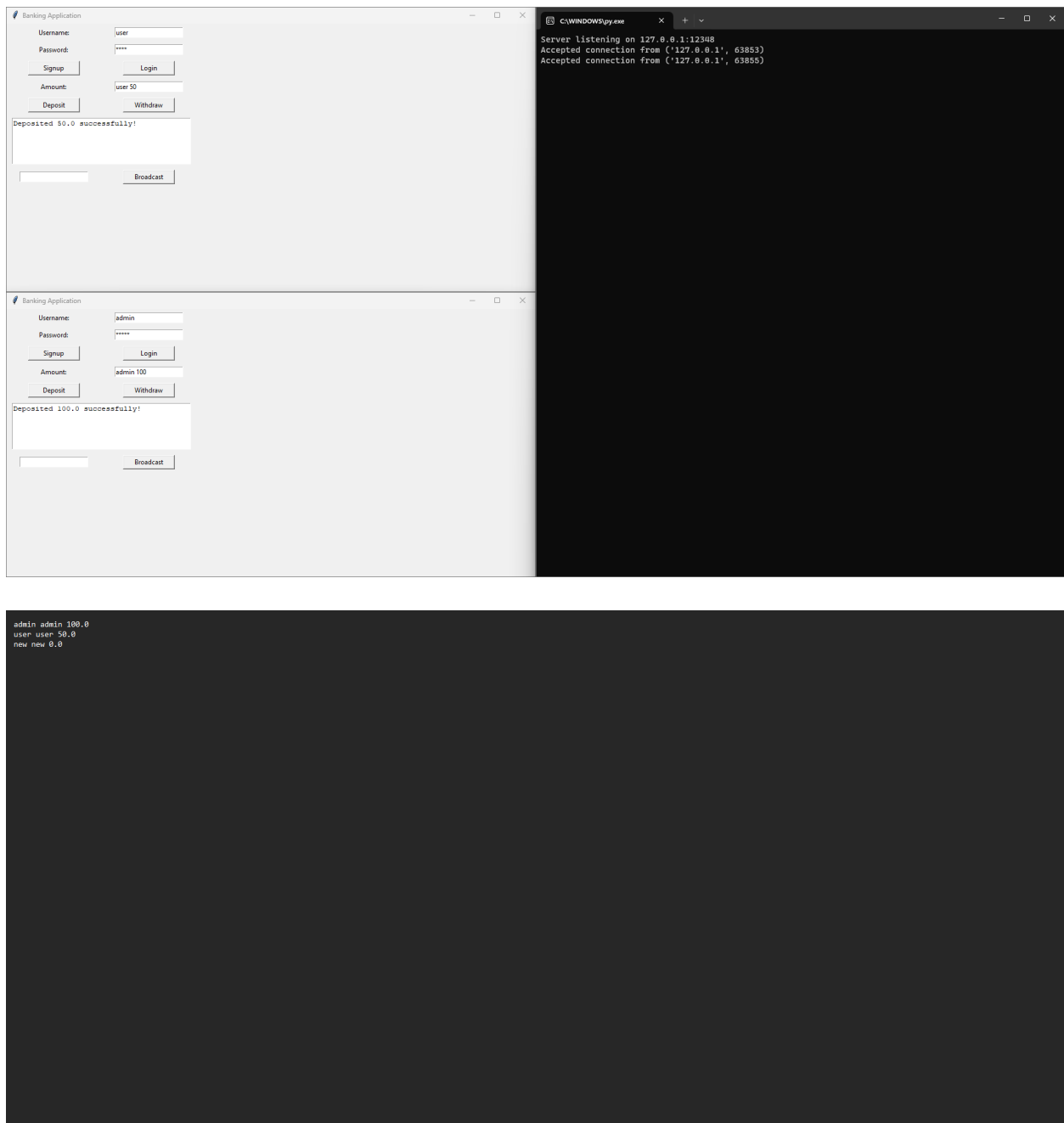
III. User Deposit and Withdraw Actions

For the following set of screenshots, the server was restarted for a clean view of the terminal.

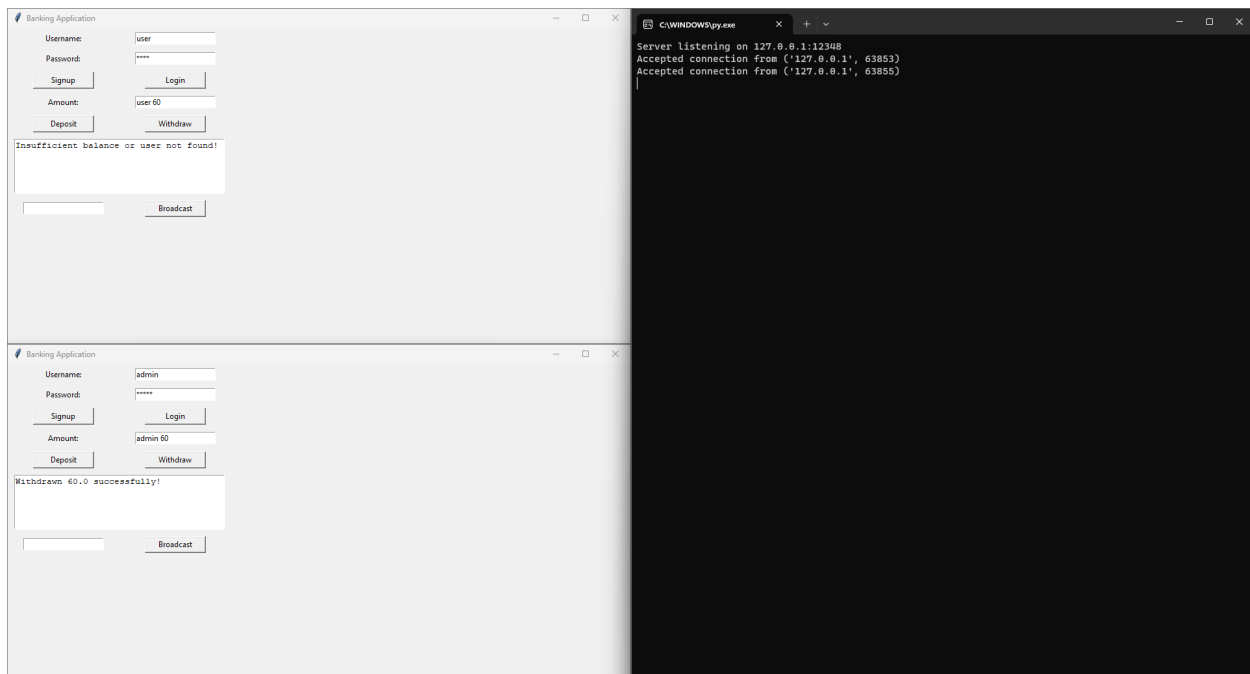


Two different clients have connected to the server simultaneously.

Each of the two connected users attempted to deposit some amount of money into their accounts. Both of their actions were successful as indicated in the GUI and the *users.txt* file.



Both users then attempt to withdraw 60 units from their respective accounts. However, since only one user had a sufficient amount, the other had received an error message indicating insufficient balance.



IV. Conclusion

This was a generally fun assignment to work on. Though simple and lacks some basic features, such as password hashing or an actual database in the backend, it successfully simulates the core architecture of a client-server application. It achieves the two goals of an always on server and the capability to handle multiple clients simultaneously.

V. References

Python *socket*: <https://docs.python.org/3/library/socket.html>

Python *tkinter*: <https://docs.python.org/3/library/tkinter.html>

Python *threading*: <https://docs.python.org/3/library/threading.html>

Python *files*: https://www.w3schools.com/python/python_file_open.asp