SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
SCHOOL OF COMPUTING
DEPARTMENT OF DATASCIENCE AND BUSINESS SYSTEMS
21CSC202J OPERATING SYSTEMS

MINI PROJECT REPORT

FILE TRANSFER USING SOCKETS IN PYTHON

Name: Ponnuri Aniruddha
Register Number: RA2112704010015
Mail ID: pp0783@srmist.edu.in
Department: MTech integrated computer science
Specialization: data Science
Semester: 3

Team Members

Name: Y Shabanya Kishore **Registration Number: RA2112704010018**

Table Of Contents

ABSTRACT:

Over the last few years, there has been a drastic change in information technology. This includes the various ways in which files can be shared and stored. Linux is a relatively a popular OS which has been steadily taking over more and more market share. Easy to use, easy to develop for, and open-source, it has picked up a following of developers who want to create content for the masses. Sockets is a popular way to create a file sharing app in Linux which can be used to share files across all platform, and with the power of python we can build an interface for the app. This paper aims to combine the two, building an socket based application for Linux/Windows, offering users the power of file sharing in the palm of their hand.

KEYWORDS: Python, Sockets, File Transfer, Tkinter

I. INTRODUCTION

File sharing is the practice of distributing or providing access to digitally stored information, such as computer programs, multimedia (audio, images and video), documents, or electronic books. It may be implemented through a variety of ways.

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library. Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0.Python 2.0 was released in 2000 and introduced new features such as list comprehensions, cycle-detecting garbage collection, reference counting, and Unicode support. Python 3.0, released in 2008, was a major revision that is not completely backward-compatible with earlier versions. Python 2 was discontinued with version 2.7.18 in 2020. Python consistently ranks as one of the most popular programming languages. Python was conceived in the late 1980s[42] by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language, which was inspired by SETL, capable of exception handling (from the start plus new capabilities in Python 3.11) and interfacing with the Amoeba operating system. Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's "benevolent dictator for life", a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. In January 2019, active Python core developers elected a five-member Steering Council to lead the project.

Python provides two levels of access to the network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols. Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on. This chapter gives you an understanding on the most famous concept in Networking - Socket Programming. Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents. Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as

well as a generic interface for handling the rest. TCP stands for Transmission Control Protocol. It is a communication protocol that is designed for end-to-end data transmission over a network. TCP is basically the ” standard” communication protocol for data transmission over the Internet. It is a highly efficient and reliable communication protocol as it uses a three-way handshake to connect the client and the server. It is a process that requires both the client and

the server to exchange synchronization (SYN) and acknowledge (ACK) packets before the data transfer takes place.

Some of the features of the TCP are as follows:

It provides end-to-end communication.

It is a connection-oriented protocol.

It provides error-checking and recovery mechanisms.

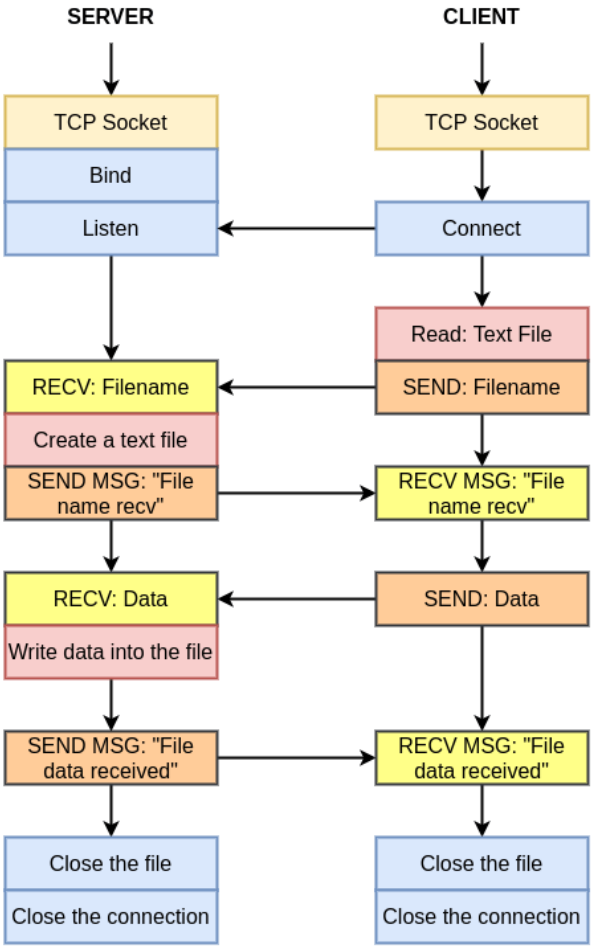
II. Review of Existing methods and their Limitations

At the moment, record structures are the only file structures that can be used directly with FTP. However, you don't have to use record structures. A user whose file doesn't have a record structure should be able to save and get his file from any HOST. If a user wants to send a record-structured file, he or she must send the right FTP "STRU" command (the default assumption is no record structure). A serving HOST doesn't have to accept record structures, but it must let the user know by sending the right response. The receiver can then get rid of any record structure information in the data stream. But there is no way to find bits that get lost or mixed up during data transfer. This problem might be best solved at the NCP level, where it will help the most people. But a restart procedure is available to protect the user from system failures, like when HOST, the FTP-process, or the IMP subnet fail. Only the block mode of data transfer has a defined restart procedure. It requires the person sending the data to put a special marker code with some marker information into the data stream. The marker information is only important to the sender, but it must be made up of ASCII characters that can be printed. The printable ASCII characters are defined as being codes 33. through 126. (This means that codes 0. through 31. and the characters SP and DEL are not printable ASCII characters.) The marker could be a bit count, a record count, or any other piece of information that could be used to find a data checkpoint. If the receiver of the data uses the restart procedure, it would mark the position of this marker in the receiving system and tell the user where it is.

Using the FTP restart procedure, the user can find the marker point and start the data transfer again if the system fails. Here are some examples of how the restart procedure can be used.

III. Proposed Method with System Architecture / Flow Diagram

The overall procedure for the TCP file transfer is presented in the figure below.



IV. Modules Description

1. Tkinter (GUI):

The tkinter package (“Tk interface”) is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, including macOS, as well as on Windows systems.

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Tkinter supports a range of Tcl/Tk versions, built either with or without thread support. The official Python binary release bundles Tcl/Tk 8.6 threaded. See the source code for the _tkinter module for more information about supported versions.

Tkinter is not a thin wrapper, but adds a fair amount of its own logic to make the experience more pythonic. This documentation will concentrate on these additions and changes, and refer to the official Tcl/Tk documentation for details that are unchanged.

2. OS:

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module, and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling see the shutil module.

Notes on the availability of these functions:

The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).

Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability.

All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.

On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.

On WebAssembly platforms wasm32-emscripten and wasm32-wasi, large parts of the os module are not available or behave differently. API related to processes (e.g. fork(), execve()),

signals (e.g. kill(), wait()), and resources (e.g. nice()) are not available. Others like getuid() and getpid() are emulated or stubs.

3. Sockets:

This module does not work or is not available on WebAssembly platforms wasm32-emscripten and wasm32-wasi. See WebAssembly platforms for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

V. IMPLEMENTATION REQUIREMENTS

Both sender and receiver should on same local network i.e. same Wi-Fi.

If you want to run the code in Terminal.

First run the main.py file in send mode

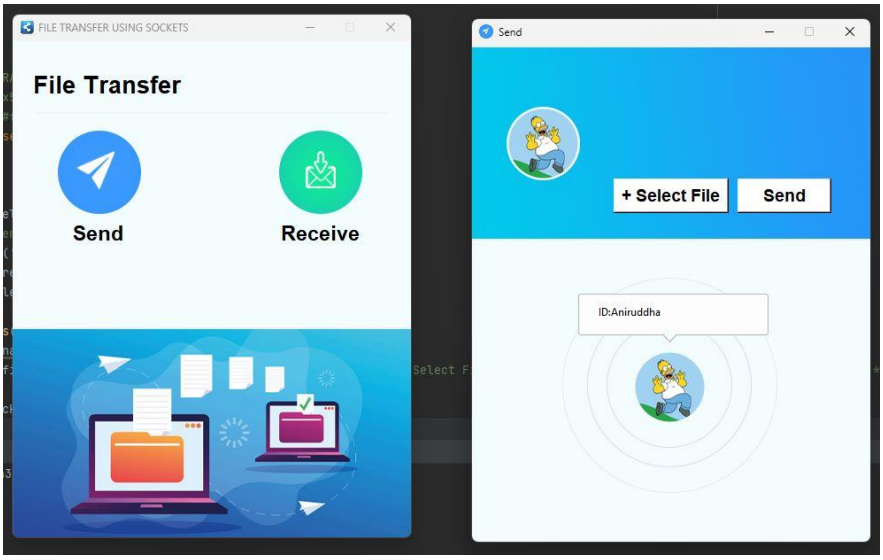
Then run the reciever.py or main.py file in receiver mode

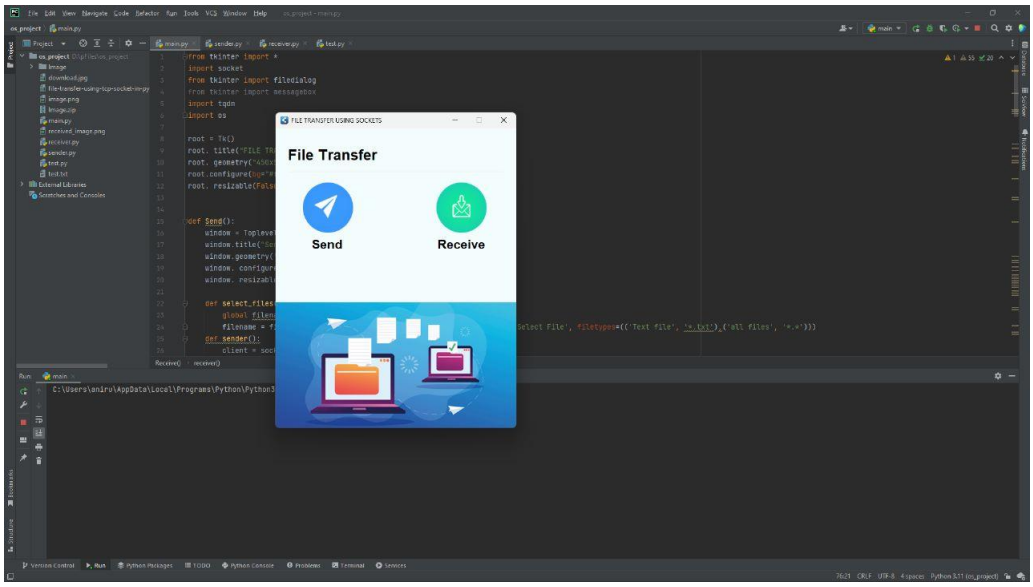
both should be running simultaneously

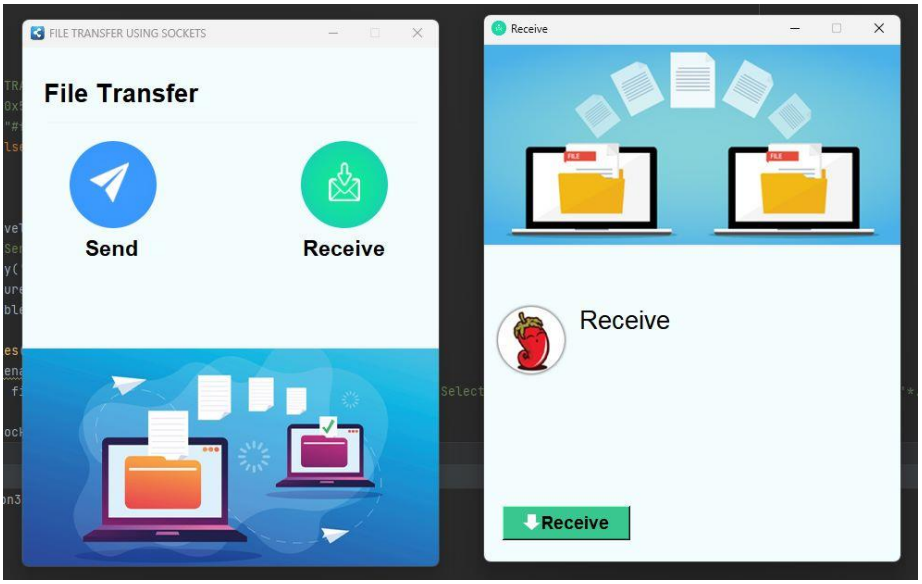
In file transfer type the filename will be "received" with file extension from sender file

After file is transfer successfully then program i.e receiver.py will exit successfully

VI. Output Screenshots







VII. CONCLUSION

This project has followed the all necessary steps to create a working file sharing app using python, from inception of the idea through to implementation.

Recommendations

There are a number of additions to the application that, despite being unnecessary in light of the previously outlined requirements, would provide the user with more functionality and increase the application's depth. Some of these are modifications to existing code to improve functionality, while others are simply the addition of new components.

Virus scanning of uploaded files would be a useful addition. Because so many files from various users would be uploaded, scanning them would provide security for all users as well as the application by preventing viruses from being shared or stored on the server.

A final thought is to provide file searching and filtering. As users begin to own and gain access to multiple boxes, it is entirely possible that they will lose track of where a specific file is, or that they will want to see a list of all files of a specific file type that they have access to. They would be able to access this information by providing some form of searching and filtering.

Of course, these are just a few extension ideas; there are many more ways to broaden the application and provide more functionality, depending on what is desired.

REFERENCES:

http://www.ijcse.net/docs/IJCSE13-02-05-055.pdf

https://www.folkstalk.com/tech/how-to-send-file-using-socket-in-python-with-code-examples/

https://www.geeksforgeeks.org/file-sharing-app-using-python/

https://www.freecodecamp.org/news/simplehttpserver-explained-how-to-send-files-using-python/

https://idiotdeveloper.com/file-transfer-using-tcp-socket-in-python3/

https://www.thepythoncode.com/article/send-receive-files-using-sockets-python

https://linuxhint.com/python_socket_file_transfer_send/

Appendix A – Source Code:

```
from tkinter import *
import socket
from tkinter import filedialog
import pathlib
import os

root = Tk()
root. title("FILE TRANSFER USING SOCKETS")
root. geometry("450x560+500+200")
root.configure(bg="#f4fdfe")
root. resizable(False, False)



def Send():
    window = Toplevel(root)
    window.title("Send",)
    window.geometry('450x560+500+200')
    window. configure(bg="#f4fdfe")
    window. resizable(False, False)

    def select_files():
        global filename
        global text
        filename = filedialog.askopenfilename(initialdir=os.getcwd(), title='Select File',
filetypes=(('Text file', '*.txt'),('all files', '*.*')))
        file_extension = pathlib.Path(filename).suffix
        text = 'received' + file_extension
    def sender():
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        client.connect(("localhost", 9999))
        file = open(filename, "rb")
        file_size = os.path.getsize(filename)
        client.send(text.encode())
        client.send(str(file_size).encode())
        data = file.read()
        client.sendall(data)
        client.send(b"<END>")
    # icon
    image_icon1 = PhotoImage(file="Image/send.png")
    window.iconphoto(False, image_icon1)
    Sbackground = PhotoImage(file="Image/sender.png")
    Label(window, image=Sbackground).place(x=-2, y=0)
    Mbackground = PhotoImage(file="Image/id.png")
    Label(window, image=Mbackground, bg="#f4fdfe").place(x=100, y=260)


    host = socket.gethostname()
    Label(window, text=f'ID:{host}', bg='white',fg='black').place(x=140, y=290)


    Button(window, text="+ Select File", width=10, height=1, font='arial 14
bold',bg="#fff", fg="#000", command=select_files).place(x=160, y=150)
    Button(window, text="Send", width=8, height=1, font='arial 14 bold',bg="#fff",
fg="#000",command=sender).place(x=300, y=150)


    window.mainloop()
```

```
def Receive():
    main = Toplevel(root)
    main.title("Receive",)
    main.geometry('450x560+500+200')
    main.configure(bg="#f4fdfe")
    main.resizable(False, False)

    def receiver():
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(("localhost", 9999))
        server.listen()

        client, addr = server.accept()
        file_name = client.recv(1024).decode()
        # print(file_name)
        file_size = client.recv(1024).decode()
        # print(file_size)
        file = open(file_name, "wb")
        file_bytes = b""
        done = False
        while not done:
            data = client.recv(1024)
            if file_bytes[-5:] == b"<END>":
                done = True
            else:
                file_bytes += data


    # icon
```

```
    image_icon1 = PhotoImage(file="Image/receive.png")
    main.iconphoto(False,image_icon1)

    Hbackground = PhotoImage(file="Image/receiver.png")
    Label(main,image=Hbackground).place(x=-2, y=0)
    logo = PhotoImage(file="Image/profile.png")
    Label(main,image=logo,bg="#f4fdfe").place(x=10,y=280)

    Label(main,text="Receive",font=('arial',20),bg="#f4fdfe").place(x=100,y=280)

    #Label(main,text="Input sender
id",font=('arial',10),bg="#f4fdfe").place(x=20,y=340)
    #SenderId=Entry(main,width=25,fg='black',border=2,bg='white',font=('arial',15))
    #SenderId.place(x=20,y=370)
    #SenderId.focus()
    #Label(main,text="Filename for the incoming file:
",font=('arial',10),bg="#f4fdfe").place(x=20,y=340)

#incoming_file=Entry(main,width=25,fg='black',border=2,bg='white',font=('arial',1
5))
    #incoming_file.place(x=20,y=450)

    imageicon=PhotoImage(file="Image/arrow.png")

r=Button(main,text="Receive",compound=LEFT,image=imageicon,width=130,bg="
#39c790",font='arial 14 bold',command=receiver)
    r.place(x=20,y=500)

    main.mainloop()
```

```
# icon

image_icon = PhotoImage(file="Image/icon.png")

root.iconphoto(False, image_icon)

Label(root, text="File Transfer", font=('Acumin Variable Concept', 20, 'bold'),
bg="#f4fdfe").place(x=20, y=31)

Frame(root, width=400, height=2, bg="#f3f5f6").place(x=25, y=80)

send_image = PhotoImage(file="Image/send.png")

send = Button(root, image=send_image, bg="#f4fdfe", bd=0, command=Send)

send.place(x=50, y=100)

receive_image = PhotoImage(file="Image/receive.png")

receive = Button(root, image=receive_image,bg="#f4fdfe", bd=0,
command=Receive)

receive.place(x=300, y=100)

# label

Label(root, text="Send", font=('Acumin Variable Concept',17, 'bold'), bg="#f4fdfe")
.place(x=65, y=200)

Label(root, text="Receive", font=('Acumin Variable Concept',17, 'bold'),
bg="#f4fdfe") .place(x=300, y=200)

background = PhotoImage(file="Image/background.png")

Label(root, image=background).place(x=-2, y=323)


root. mainloop()
```

Appendix B – GITHUB PROFILE AND SOUCRE CODE

https://github.com/RA2112704010015

https://github.com/RA2112704010015/OS_MINI_PROJECT