# Assignment - QR File Share on network

File transfer is the process of copying or moving a file from one computer to another over a network or Internet connection. In this tutorial, we’ll go step by step on how you can write client/server Python scripts that handles that.

The basic idea is to create a server that listens on a particular port; this server will be responsible for receiving files (you can make the server send files as well). On the other hand, the client will try to connect to the server and send a file of any type.

We will use the [socket](https://docs.python.org/3/library/socket.html) module, which comes built-in with Python and provides us with socket operations that are widely used on the Internet, as they are behind any connection to any network.

Please note that there are more reliable ways to transfer files with tools like rsync or scp. However, the goal of this tutorial is to transfer files with Python programming language and without any third-party tool.

First, we gonna need to install the ***tqdm*** library, which will enable us to print fancy progress bars:

pip3 install tqdm

## Client Code

Let’s start with the client, the sender:

import socket  
import tqdm  
import os  
  
SEPARATOR = "<SEPARATOR>"  
BUFFER\_SIZE = 4096 # send 4096 bytes each time step

We need to specify the IP address, the port of the server we want to connect to, and the name of the file we want to send.

# the ip address or hostname of the server, the receiver  
host = "192.168.1.101"  
# the port, let's use 5001  
port = 5001  
# the name of file we want to send, make sure it exists  
filename = "data.csv"  
# get the file size  
filesize = os.path.getsize(filename)

The filename needs to exist in the current directory, or you can use an absolute path to that file somewhere on your computer. This is the file you want to send.

os.path.getsize(filename) gets the size of that file in bytes; that’s great, as we need it for printing progress bars in the client and the server.

Let’s create the TCP socket:

# create the client socket  
s = socket.socket()

Connecting to the server:

print(f"[+] Connecting to {host}:{port}")  
s.connect((host, port))  
print("[+] Connected.")

connect() method expects an address of the pair (host, port) to connect the socket to that remote address. Once the connection is established, we need to send the name and size of the file:

# send the filename and filesize  
s.send(f"{filename}{SEPARATOR}{filesize}".encode())

I’ve used SEPARATOR here to separate the data fields; it is just a junk message, we can just use send() twice, but we may not want to do that anyway. encode() function encodes the string we passed to ‘utf-8’ encoding (that’s necessary).

Now we need to send the file, and as we are sending the file, we’ll print nice progress bars using the tqdm library:

# start sending the file  
progress = tqdm.tqdm(range(filesize), f"Sending {filename}", unit="B", unit\_scale=True, unit\_divisor=1024)  
with open(filename, "rb") as f:  
 while True:  
 # read the bytes from the file  
 bytes\_read = f.read(BUFFER\_SIZE)  
 if not bytes\_read:  
 # file transmitting is done  
 break  
 # we use sendall to assure transimission in   
 # busy networks  
 s.sendall(bytes\_read)  
 # update the progress bar  
 progress.update(len(bytes\_read))  
# close the socket  
s.close()

Basically, what we are doing here is opening the file as read in binary, read chunks from the file (in this case, 4096 bytes or 4KB) and sending them to the socket using the sendall() function, and then we update the progress bar each time, once that’s finished, we close that socket.

## Server Code

Alright, so we are done with the client. Let’s dive into the server, so open up a new empty Python file and:

import socket  
import tqdm  
import os  
# device's IP address  
SERVER\_HOST = "0.0.0.0"  
SERVER\_PORT = 5001  
# receive 4096 bytes each time  
BUFFER\_SIZE = 4096  
SEPARATOR = "<SEPARATOR>"

I’ve initialized some parameters we are going to use. Notice that I’ve used “0.0.0.0” as the server IP address. This means all IPv4 addresses that are on the local machine. You may wonder why we don’t just use our local IP address or “localhost” or “127.0.0.1”? Well, if the server has two IP addresses, let’s say “192.168.1.101” on a network and “10.0.1.1” on another, and the server listens on “0.0.0.0”, it will be reachable at both of those IPs.

Alternatively, you can use your public or private IP address, depending on your clients. If the connected clients are in your local network, you should use your private IP (you can check it using ipconfig command in Windows or ifconfig command in Mac OS/Linux), but if you’re expecting clients from the Internet, you definitely should use your public address.

Also, Make sure you use the same port in the server as in the client.

Let’s create our TCP socket:

# create the server socket  
# TCP socket  
s = socket.socket()

Now, this is different from the client, and we need to bind the socket we just created to our SERVER\_HOST and SERVER\_PORT:

# bind the socket to our local address  
s.bind((SERVER\_HOST, SERVER\_PORT))

After that, we are going to listen for connections:

# enabling our server to accept connections  
# 5 here is the number of unaccepted connections that  
# the system will allow before refusing new connections  
s.listen(5)  
print(f"[\*] Listening as {SERVER\_HOST}:{SERVER\_PORT}")

Once the client connects to our server, we need to accept that connection:

# accept connection if there is any  
client\_socket, address = s.accept()   
# if below code is executed, that means the sender is connected  
print(f"[+] {address} is connected.")

Remember that when the client is connected, it’ll send the name and size of the file. Let’s receive them:

# receive the file infos  
# receive using client socket, not server socket  
received = client\_socket.recv(BUFFER\_SIZE).decode()  
filename, filesize = received.split(SEPARATOR)  
# remove absolute path if there is  
filename = os.path.basename(filename)  
# convert to integer  
filesize = int(filesize)

As mentioned earlier, the received data is combined with the filename and the filesize, and we can easily extract them by splitting them by the SEPARATOR string.

After that, we need to remove the file’s absolute path because the sender sent the file with his own file path, which may differ from ours, the os.path.basename() function returns the final component of a path name.

Now we need to receive the file:

# start receiving the file from the socket  
# and writing to the file stream  
progress = tqdm.tqdm(range(filesize), f"Receiving {filename}", unit="B", unit\_scale=True, unit\_divisor=1024)  
with open(filename, "wb") as f:  
 while True:  
 # read 1024 bytes from the socket (receive)  
 bytes\_read = client\_socket.recv(BUFFER\_SIZE)  
 if not bytes\_read:   
 # nothing is received  
 # file transmitting is done  
 break  
 # write to the file the bytes we just received  
 f.write(bytes\_read)  
 # update the progress bar  
 progress.update(len(bytes\_read))  
  
# close the client socket  
client\_socket.close()  
# close the server socket  
s.close()

Not entirely different from the client code. However, we are opening the file as write in binary here and using the recv(BUFFER\_SIZE) method to receive BUFFER\_SIZE bytes from the client socket and write it to the file. Once that’s finished, we close both the client and server sockets.

## Testing the code

\_**Learn also:** How to List all Files and Directories in FTP Server using Python

Alright, let me try it on my own private network:

C:\> python receiver.py  
  
[\*] Listening as 0.0.0.0:5001

I need to go to my Linux box and send an example file:

yilmaz@intec:~/tools# python3 sender.py  
[+] Connecting to 192.168.1.101:5001  
[+] Connected.  
Sending data.npy: 9%|███████▊ | 45.5M/487M [00:14<02:01, 3.80MB/s]

Let’s see the server now:

[+] ('192.168.1.101', 47618) is connected.  
Receiving data.npy: 33%|███████████████████▍ | 160M/487M [01:04<04:15, 1.34MB/s]

Great, we are done!

You can extend this code for your own needs now. Here are some examples you can implement:

* Enabling the server to receive multiple files from multiple clients simultaneously using threads.
* Compressing the files before sending them which may help increase the transfer duration. If the target files you want to send are images, you can optimize images by compressing them.
* Encrypting the file before sending it to ensure that no one can intercept and read that file.
* Ensuring the file is appropriately sent by checking the checksums of both files (the original file of the sender and the sent file in the receiver). In this case, you need secure hashing algorithms to do it.
* Adding a chat room so you can both chat and transfer files.

Good luck!