****

**Server-Browser Project**

Team Members

Mohammed AlHabbash  (120170699)

Abdallah Abo Abdo      (120170526)

Hassan Sammour          (120170878)

Abdullah Ouda               (12016 0453)

Usama Al Zayan                         (120171073)

prof. Aiman Abu Samra

**project Link**

<https://github.com/Abdallah-Abo-Abdo/Server-Browser-Project.git>

## **What is a Socket?**

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

## **Where is Socket Used?**

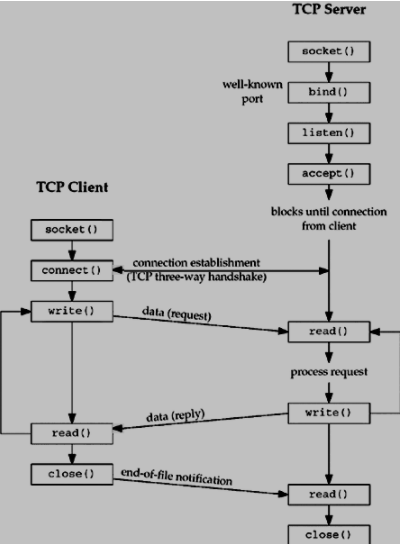
A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

## **Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

* **Stream Sockets.**
* **Datagram Sockets.**
* **Raw Sockets.**
* **Sequenced Packet Sockets.**



## **Requirement:**

1. Implement one-way data transmission. One sends data, and the other receives data.
2. Implement Client and Server send and receive data at the same time.
3. Try to transmit a media file and analyze the features of TCP/UDP.

Implement number 1:

Simple code consists form server and client to implement on-way transmit data:

Server code:

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind((socket.gethostname(), 1234))

s.listen(5)

while True:

    # now our endpoint knows about the OTHER endpoint.

    clientsocket, address = s.accept()

    print(f"Connection from {address} has been established.")

    clientsocket.send(bytes("Wellcom To Server!!!","utf-8"))

    clientsocket.close()

Client Code:

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.connect((socket.gethostname(), 1234))

while True:

    full\_msg = ''

    while True:

        msg = s.recv(8)

        if len(msg) <= 0:

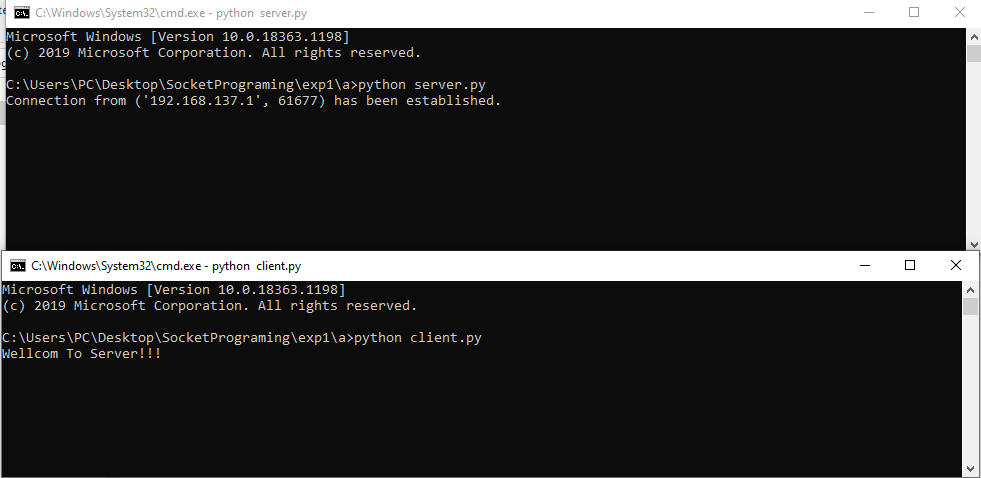
            break

        full\_msg += msg.decode("utf-8")

    if len(full\_msg) > 0:

        print(full\_msg)

Run both scripts Client and Server :



Implement number 2:

Server code deal with multi clients send and receive in same time

import socket

import select

HEADER\_LENGTH = 10

IP = "127.0.0.1"

PORT = 1234

# Create a socket

# socket.AF\_INET - address family, IPv4, some otehr possible are AF\_INET6, AF\_BLUETOOTH, AF\_UNIX

# socket.SOCK\_STREAM - TCP, conection-based, socket.SOCK\_DGRAM - UDP, connectionless, datagrams, socket.SOCK\_RAW - raw IP packets

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# SO\_ - socket option

# SOL\_ - socket option level

# Sets REUSEADDR (as a socket option) to 1 on socket

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

# Bind, so server informs operating system that it's going to use given IP and port

# For a server using 0.0.0.0 means to listen on all available interfaces, useful to connect locally to 127.0.0.1 and remotely to LAN interface IP

server\_socket.bind((IP, PORT))

# This makes server listen to new connections

server\_socket.listen()

# List of sockets for select.select()

sockets\_list = [server\_socket]

# List of connected clients - socket as a key, user header and name as data

clients = {}

print(f'Listening for connections on {IP}:{PORT}...')

# Handles message receiving

def receive\_message(client\_socket):

    try:

        # Receive our "header" containing message length, it's size is defined and constant

        message\_header = client\_socket.recv(HEADER\_LENGTH)

        # If we received no data, client gracefully closed a connection, for example using socket.close() or socket.shutdown(socket.SHUT\_RDWR)

        if not len(message\_header):

            return False

        # Convert header to int value

        message\_length = int(message\_header.decode('utf-8').strip())

        # Return an object of message header and message data

        return {'header': message\_header, 'data': client\_socket.recv(message\_length)}

    except:

        # If we are here, client closed connection violently, for example by pressing ctrl+c on his script

        # or just lost his connection

        # socket.close() also invokes socket.shutdown(socket.SHUT\_RDWR) what sends information about closing the socket (shutdown read/write)

        # and that's also a cause when we receive an empty message

        return False

while True:

    # Calls Unix select() system call or Windows select() WinSock call with three parameters:

    #   - rlist - sockets to be monitored for incoming data

    #   - wlist - sockets for data to be send to (checks if for example buffers are not full and socket is ready to send some data)

    #   - xlist - sockets to be monitored for exceptions (we want to monitor all sockets for errors, so we can use rlist)

    # Returns lists:

    #   - reading - sockets we received some data on (that way we don't have to check sockets manually)

    #   - writing - sockets ready for data to be send thru them

    #   - errors  - sockets with some exceptions

    # This is a blocking call, code execution will "wait" here and "get" notified in case any action should be taken

    read\_sockets, \_, exception\_sockets = select.select(sockets\_list, [], sockets\_list)

    # Iterate over notified sockets

    for notified\_socket in read\_sockets:

        # If notified socket is a server socket - new connection, accept it

        if notified\_socket == server\_socket:

            # Accept new connection

            # That gives us new socket - client socket, connected to this given client only, it's unique for that client

            # The other returned object is ip/port set

            client\_socket, client\_address = server\_socket.accept()

            # Client should send his name right away, receive it

            user = receive\_message(client\_socket)

            # If False - client disconnected before he sent his name

            if user is False:

                continue

            # Add accepted socket to select.select() list

            sockets\_list.append(client\_socket)

            # Also save username and username header

            clients[client\_socket] = user

            print('Accepted new connection from {}:{}, username: {}'.format(\*client\_address, user['data'].decode('utf-8')))

        # Else existing socket is sending a message

        else:

            # Receive message

            message = receive\_message(notified\_socket)

            # If False, client disconnected, cleanup

            if message is False:

                print('Closed connection from: {}'.format(clients[notified\_socket]['data'].decode('utf-8')))

                # Remove from list for socket.socket()

                sockets\_list.remove(notified\_socket)

                # Remove from our list of users

                del clients[notified\_socket]

                continue

            # Get user by notified socket, so we will know who sent the message

            user = clients[notified\_socket]

            print(f'Received message from {user["data"].decode("utf-8")}: {message["data"].decode("utf-8")}')

            # Iterate over connected clients and broadcast message

            for client\_socket in clients:

                # But don't sent it to sender

                if client\_socket != notified\_socket:

                    # Send user and message (both with their headers)

                    # We are reusing here message header sent by sender, and saved username header send by user when he connected

                    client\_socket.send(user['header'] + user['data'] + message['header'] + message['data'])

    # It's not really necessary to have this, but will handle some socket exceptions just in case

    for notified\_socket in exception\_sockets:

        # Remove from list for socket.socket()

        sockets\_list.remove(notified\_socket)

        # Remove from our list of users

        del clients[notified\_socket]

.

Client Code:

import socket

import select

import errno

HEADER\_LENGTH = 10

IP = "127.0.0.1"

PORT = 1234

my\_username = input("Username: ")

# Create a socket

# socket.AF\_INET - address family, IPv4, some otehr possible are AF\_INET6, AF\_BLUETOOTH, AF\_UNIX

# socket.SOCK\_STREAM - TCP, conection-based, socket.SOCK\_DGRAM - UDP, connectionless, datagrams, socket.SOCK\_RAW - raw IP packets

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# Connect to a given ip and port

client\_socket.connect((IP, PORT))

# Set connection to non-blocking state, so .recv() call won;t block, just return some exception we'll handle

client\_socket.setblocking(False)

# Prepare username and header and send them

# We need to encode username to bytes, then count number of bytes and prepare header of fixed size, that we encode to bytes as well

username = my\_username.encode('utf-8')

username\_header = f"{len(username):<{HEADER\_LENGTH}}".encode('utf-8')

client\_socket.send(username\_header + username)

while True:

    # Wait for user to input a message

    message = input(f'{my\_username} > ')

    # If message is not empty - send it

    if message:

        # Encode message to bytes, prepare header and convert to bytes, like for username above, then send

        message = message.encode('utf-8')

        message\_header = f"{len(message):<{HEADER\_LENGTH}}".encode('utf-8')

        client\_socket.send(message\_header + message)

    try:

        # Now we want to loop over received messages (there might be more than one) and print them

        while True:

            # Receive our "header" containing username length, it's size is defined and constant

            username\_header = client\_socket.recv(HEADER\_LENGTH)

            # If we received no data, server gracefully closed a connection, for example using socket.close() or socket.shutdown(socket.SHUT\_RDWR)

            if not len(username\_header):

                print('Connection closed by the server')

                sys.exit()

            # Convert header to int value

            username\_length = int(username\_header.decode('utf-8').strip())

            # Receive and decode username

            username = client\_socket.recv(username\_length).decode('utf-8')

            # Now do the same for message (as we received username, we received whole message, there's no need to check if it has any length)

            message\_header = client\_socket.recv(HEADER\_LENGTH)

            message\_length = int(message\_header.decode('utf-8').strip())

            message = client\_socket.recv(message\_length).decode('utf-8')

            # Print message

            print(f'{username} > {message}')

    except IOError as e:

        # This is normal on non blocking connections - when there are no incoming data error is going to be raised

        # Some operating systems will indicate that using AGAIN, and some using WOULDBLOCK error code

        # We are going to check for both - if one of them - that's expected, means no incoming data, continue as normal

        # If we got different error code - something happened

        if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:

            print('Reading error: {}'.format(str(e)))

            sys.exit()

        # We just did not receive anything

        continue

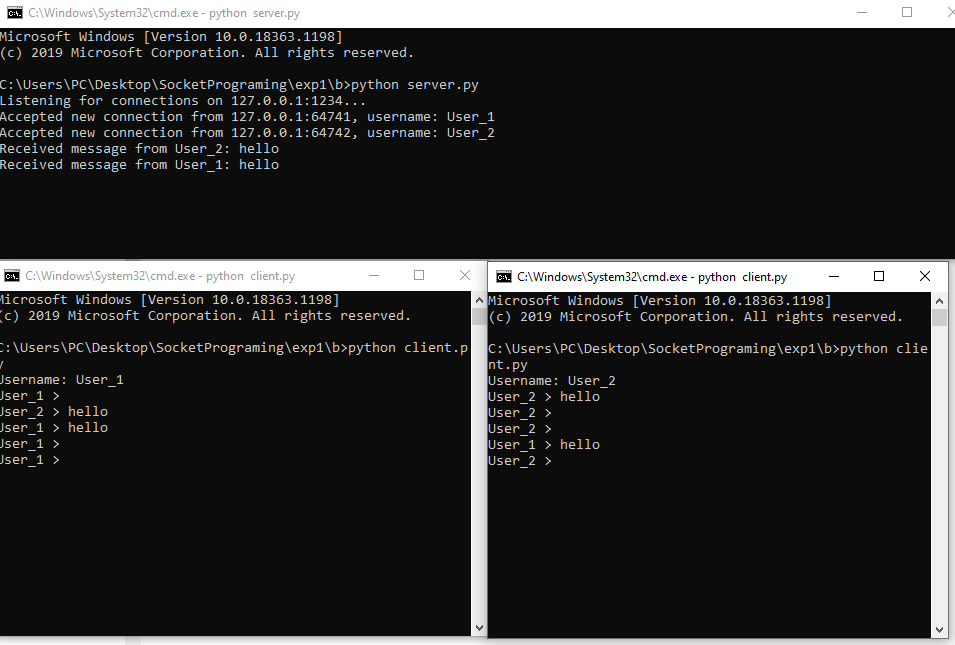
    except Exception as e:

        # Any other exception - something happened, exit

        print('Reading error: '.format(str(e)))

        sys.exit()

result:



Implement number 3:

Send files through socket from server to client

Server code

import socket

import sys

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind((socket.gethostname(), 9999))

s.listen(5)

print("Listening ...")

while True:

    conn, addr = s.accept()

    print("[+] Client connected: ", addr)

    # get file name to download

    f = open("newmytext.txt", "wb")

    while True:

        # get file bytes

        data = conn.recv(4096)

        if not data:

            break

        # write bytes on file

        f.write(data)

    f.close()

    print("[+] Download complete!")

    # close connection

    conn.close()

    print("[-] Client disconnected")

    sys.exit(0)

Client code:

import socket

import sys

HOST = "192.168.1.100"

PORT = 9999

s = socket.socket(socket.AF\_INET,   socket.SOCK\_STREAM)

s.connect((socket.gethostname(), 9999))

print("[+] Connected with Server")

# get file name to send

f\_send = "mytext.txt"

# open file

with open(f\_send, "rb") as f:

    # send file

    print("[+] Sending file...")

    data = f.read()

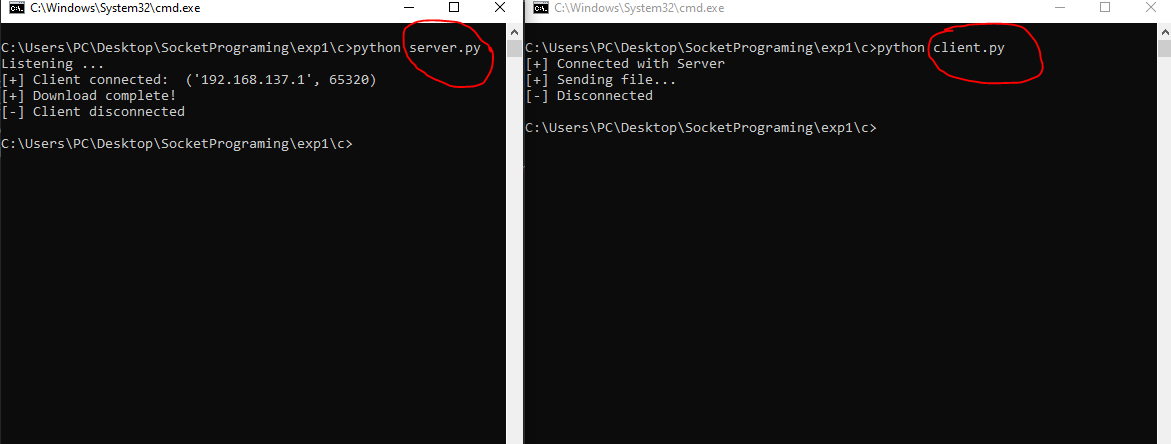
    s.sendall(data)

    # close connection

    s.close()

    print("[-] Disconnected")

    sys.exit(0)



Features of TCP/UDP:

