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?

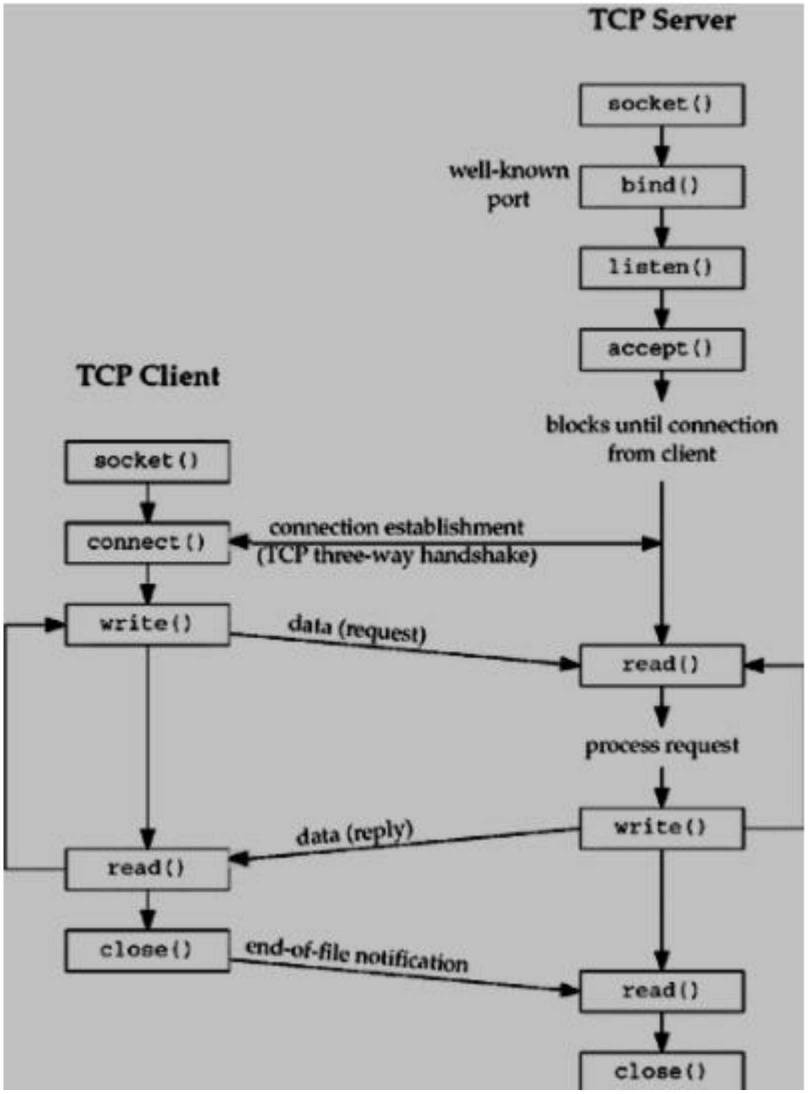
A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

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:

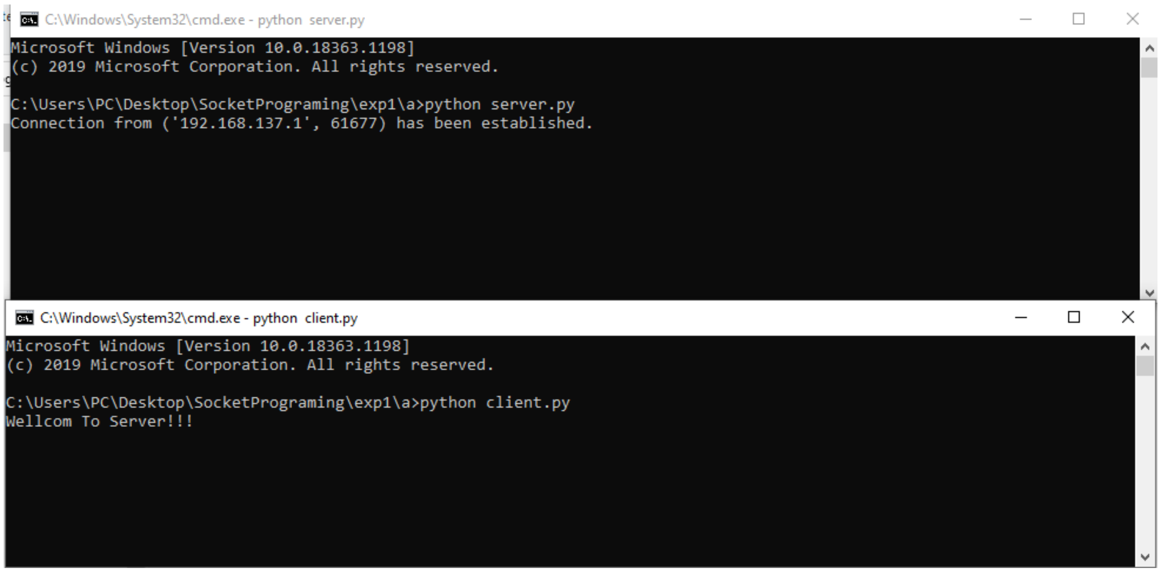
Server code:

```
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 dat
a
clients = {}

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

```python
# Handles message receiving
def receive_message(client_socket):

    try:

        # Receive our "header" containing message length, it's size is def
ined and constant
        message_header = client_socket.recv(HEADER_LENGTH)

        # If we received no data, client gracefully closed a connection, f
or 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(messa
ge_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) wh
at 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 wit
h three parameters:
    #   - rlist - sockets to be monitored for incoming data
    #   - wlist - sockets for data to be send to (checks if for example bu
ffers are not full and socket is ready to send some data)
    #   - xlist - sockets to be monitored for exceptions (we want to monit
or all sockets for errors, so we can use rlist)
    # Returns lists:
    #   - reading - sockets we received some data on (that way we don't ha
ve 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" n
otified in case any action should be taken
    read_sockets, _, exception_sockets = select.select(sockets_list, [], s
ockets_list)


    # Iterate over notified sockets
    for notified_socket in read_sockets:

        # If notified socket is a server socket - new connection, accept i
t
        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: {}'.forma
t(*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 me
ssage
            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, a
nd saved username header send by user when he connected
                    client_socket.send(user['header'] + user['data'] + mes
sage['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]
```

.

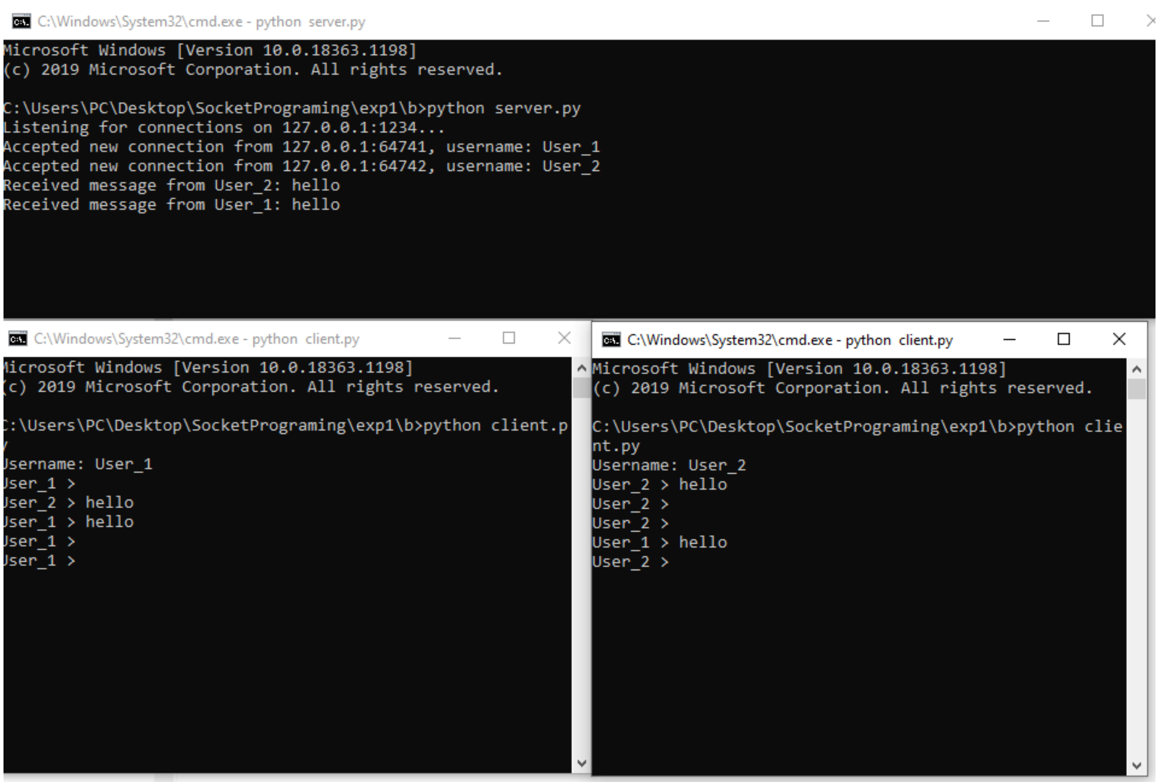
Client Code:

```
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 prep
are 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, li
ke 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 i
s defined and constant
            username_header = client_socket.recv(HEADER_LENGTH)

            # If we received no data, server gracefully closed a connectio
n, 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 rec
eived 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 i
ncoming 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 expecte
d, means no incoming data, continue as normal
        # If we got different error code - something happened
```

```python
        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:

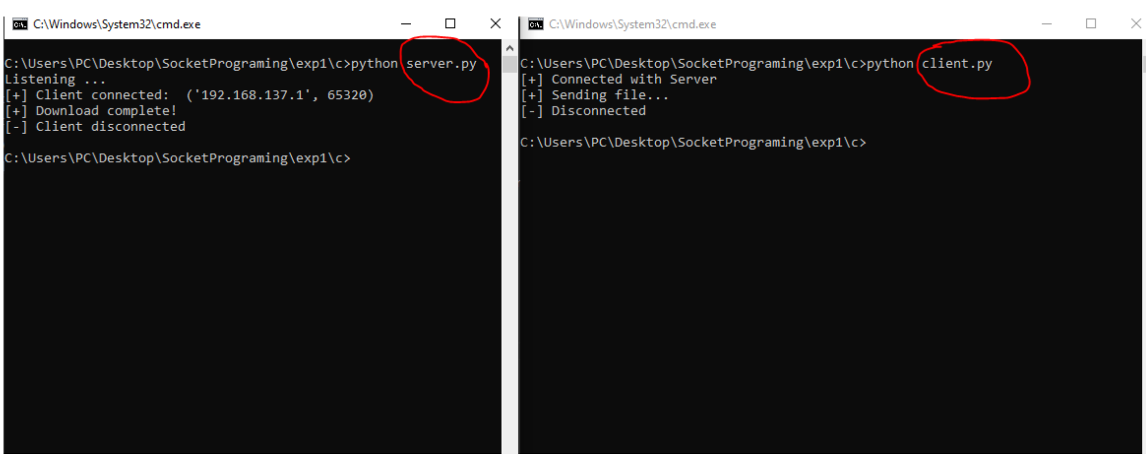
Send files through socket from server to client

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:

Summary Comparison of UDP and TCP		
Characteristic / Description	**UDP**	**TCP**
General Description	Simple, high-speed, low-functionality "wrapper" that interfaces applications to the network layer and does little else.	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues.
Protocol Connection Setup	Connectionless; data is sent without setup.	Connection-oriented; connection must be established prior to transmission.
Data Interface To Application	Message-based; data is sent in discrete packages by the application.	Stream-based; data is sent by the application with no particular structure.
Reliability and Acknowledgments	Unreliable, best-effort delivery without acknowledgments.	Reliable delivery of messages; all data is acknowledged.
Retransmissions	Not performed. Application must detect lost data and retransmit if needed.	Delivery of all data is managed, and lost data is retransmitted automatically.
Features Provided to Manage Flow of Data	None	Flow control using sliding windows; window size adjustment heuristics; congestion avoidance algorithms.
Overhead	Very low	Low, but higher than UDP
Transmission Speed	Very high	High, but not as high as UDP
Data Quantity Suitability	Small to moderate amounts of data (up to a few hundred bytes)	Small to very large amounts of data (up to gigabytes)
Types of Applications That Use The Protocol	Applications where data delivery speed matters more than completeness, where	Most protocols and applications sending data that must be