Socket Programming – Lab3

A. Murad, D. Palma

Contents

Introduction

The goal of this lab is to learn TCP and UDP socket programming by building a simple network application program (chat application), as well as analyzing TCP and UDP connections. The lab has several **milestones**. Make sure you reach each one before advancing to the next.

For delivery, submit a PDF report where you answer **only** those steps that are marked with **REPORT(%)**. Additionally, you should submit the codes or capture files, if they were **explicitly** asked for. The percent point gives you an indication of the score of that question. Lab3 counts for **4 points** of your final score in this course.

System Setup

In this lab, you will build a chatroom using a client-server architecture with TCP sockets, and a peer-to-peer (p2p) chat with UDP sockets, as shown in Figure 1. The overall procedure is for clients connect to the server via TCP and register with their name. The server broadcasts information of new clients as well as public messages to all connected clients. Clients can also chat with each other using direct UDP connections.

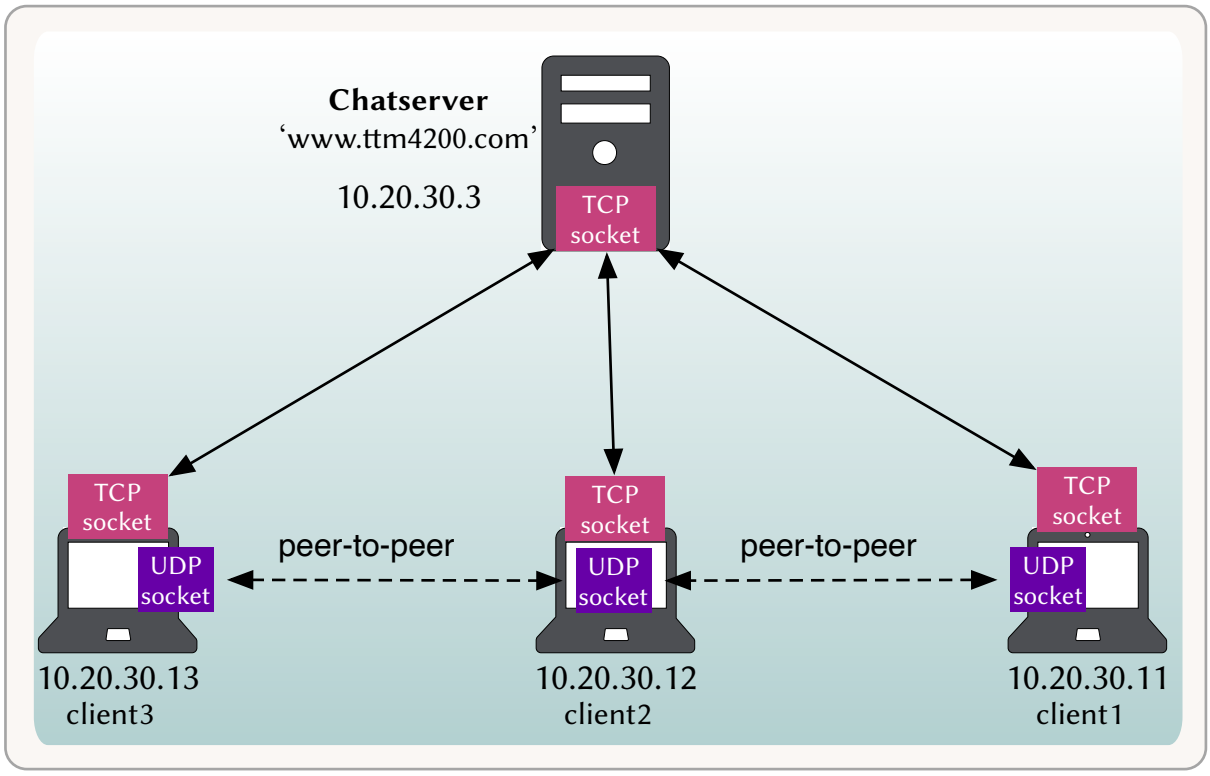

Figure 1: System Setup: chat application with socket programming

- **Use the same docker files as in lab2. You can run your chat server application in the "web-server" container, and the chat client in "client1", "client2", and "client3" containers**
- You can write the code in your machine using your favorite python IDE, then copy the code files to the container either using `scp` or the mounted volumes.
- To get an overview of using sockets in python, read this Python HOWTO (https://docs.python.org/3/howto/sockets.html), which discuss TCP sockets, and (https://docs.python.org/3/howto/sockets.html), which discuss UDP sockets. Additionally, you can read section 2.7 of the book.
- We will also use threading to enable running multiple sockets concurrently. You can have a look at this tutorial (https://dzone.com/articles/python-thread-part-1), to get a brief introduction into python threads.
- Finally, we will use *json* for serializing data (TTM4175 (https://ttm4175.iik.ntnu.no/prep-iot-http-json.html)).

1 Milestone 1 – Building Chat Server Application

In this milestone, you will build a chat server using the skeleton code below and the workflow as shown in figure 2. You are to complete the skeleton code. You can find the python file of the skeleton code in "chatserver.py" The places where you need to fill in code are marked with `====fill in here====`.

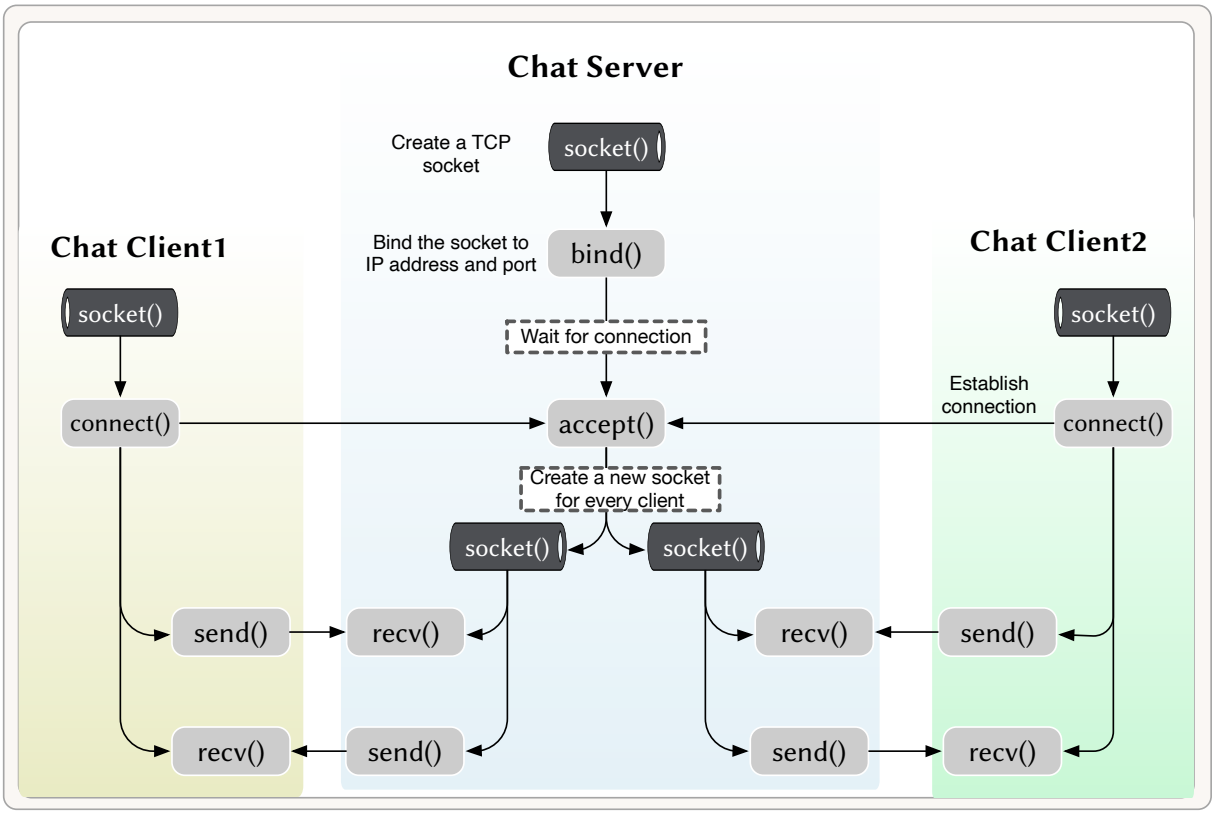

Figure 2: Chat pplication workflow: Client-server architecture using TCP sockets in python

```python
# Import necessary modules
import socket
from threading import Thread
import json
import sys

# Get the IP address of the server using socket
# Don't write the IP address directly.
SERVER_IP  = #====fill in here====

# We will use a fixed port number
PORT = 5000

# buffer size for receiving messages
RECV_BUFFER = 1024
# Dictionary for storing active Users
ACTIVE_USERS ={}  # Maps the name to the IP address

# Dictionary for storing active sockets
SOCKETS = {}  # Maps the socket to the name

# Create a server socket (TCP socket)
SERVER_SOCKET = #====fill in here====

# bind the "SERVER_SOCKET" to the SERVER_IP and PORT
# ====fill in here====

# Main thread: accepting connections from clients and making
# a child thread for every client
def accept_incoming_connections():
    while True:
        # make the SERVER_SOCKET accept  connection from a client
        client_socket, client_address = #====fill in here====
        try:
            #recieve the first message from the client_socket
            name = #====fill in here====
            name= name.decode("utf-8")
            #Storing the client name and its socket
            SOCKETS[client_socket] = name
            ACTIVE_USERS[name] = client_address[0]
            print(f"{name} with IP address:{client_address[0]}, has connected\n")
            # Updating all clients of the current active users
            broadcast_active_users()

        except:
            # close the client_socket if an error occurs
            client_socket.close()
            print('Error connecting to a client')

        #Starting a thread for the connected client
        Thread(target=handle_client, args=(client_socket,)).start()
```

```
    #Thread for handling connection to every client
    def handle_client(client_socket):
        while True:
            try:
                # receive a message from the client_socket
                msg = #====fill in here====
                msg = msg.decode("utf-8")

            except:
                # handle broken socket connection (e.g. the client pressed ctrl+c)
                client_socket.close()
                del(ACTIVE_USERS[SOCKETS[client_socket]])
                del SOCKETS[client_socket]
                broadcast_active_users()
                sys.exit()

            # len(msg)==0 when exiting wiht KeyboardInterrupt (ctrl+c)
            if len(msg)==0:
                client_socket.close()
                del(ACTIVE_USERS[SOCKETS[client_socket]])
                del(SOCKETS[client_socket])
                broadcast_active_users()
                sys.exit()
            else:
                # Attaching the name of sending client to msg
                msg = f"{SOCKETS[client_socket]}: {msg} "

                # Send the received msg to every client, except this client
                #====fill in here====


    # Function to send a message to every client, except the sending client
    def broadcast_msg(sending_client,msg):
        msg = msg.encode("utf-8")
        for client_socket in list(SOCKETS):
            # exclude sending msg to the sending_client
            if client_socket !=  sending_client:
                try:
                    # Send the msg to client_socket
                    #====fill in here====

                except:
                    client_socket.close()
                    del(ACTIVE_USERS[SOCKETS[client_socket]])
                    del (SOCKETS[client_socket])
                    broadcast_active_users()


    # Function to broadcast the current active users to all clients
    def broadcast_active_users():
        active_users ='!'+json.dumps(ACTIVE_USERS)
        active_users = active_users.encode("utf-8")
        for client_socket in list(SOCKETS):
            try:
```

```
                # send "active_users" to client_socket
                #====fill in here====

            except:
                client_socket.close()
                del(ACTIVE_USERS[SOCKETS[client_socket]])
                del (SOCKETS[client_socket])


    if __name__ == "__main__":
        # Make the server listen for TCP connection requests
        # (5 maximum number of queued connections)
        SERVER_SOCKET.listen(5)
        print("Waiting for connection...")
        # starting the main thread and joining all child threads
        connection_threads = Thread(target=accept_incoming_connections)
        connection_threads.start()
        connection_threads.join()
        # after existing from the main thread, close SERVER_SOCKET
        SERVER_SOCKET.close()
```

2 Milestone 2 – Building Chat Client Application

In this milestone, you will build a chat client using the skeleton code below. You are to complete the skeleton code. You can find the python file of the skeleton code in "chatclient.py" The places where you need to fill in code are marked with `====fill in here====`.

In the client application, a connection is first established to the chat server using a TCP socket (figure 2) and the client name is registered. Then the application will get a dictionary of all active user names and their IP addresses. In this way, the client can chat with another client (p2p) using UDP sockets, as shown in the workflow figure 3.

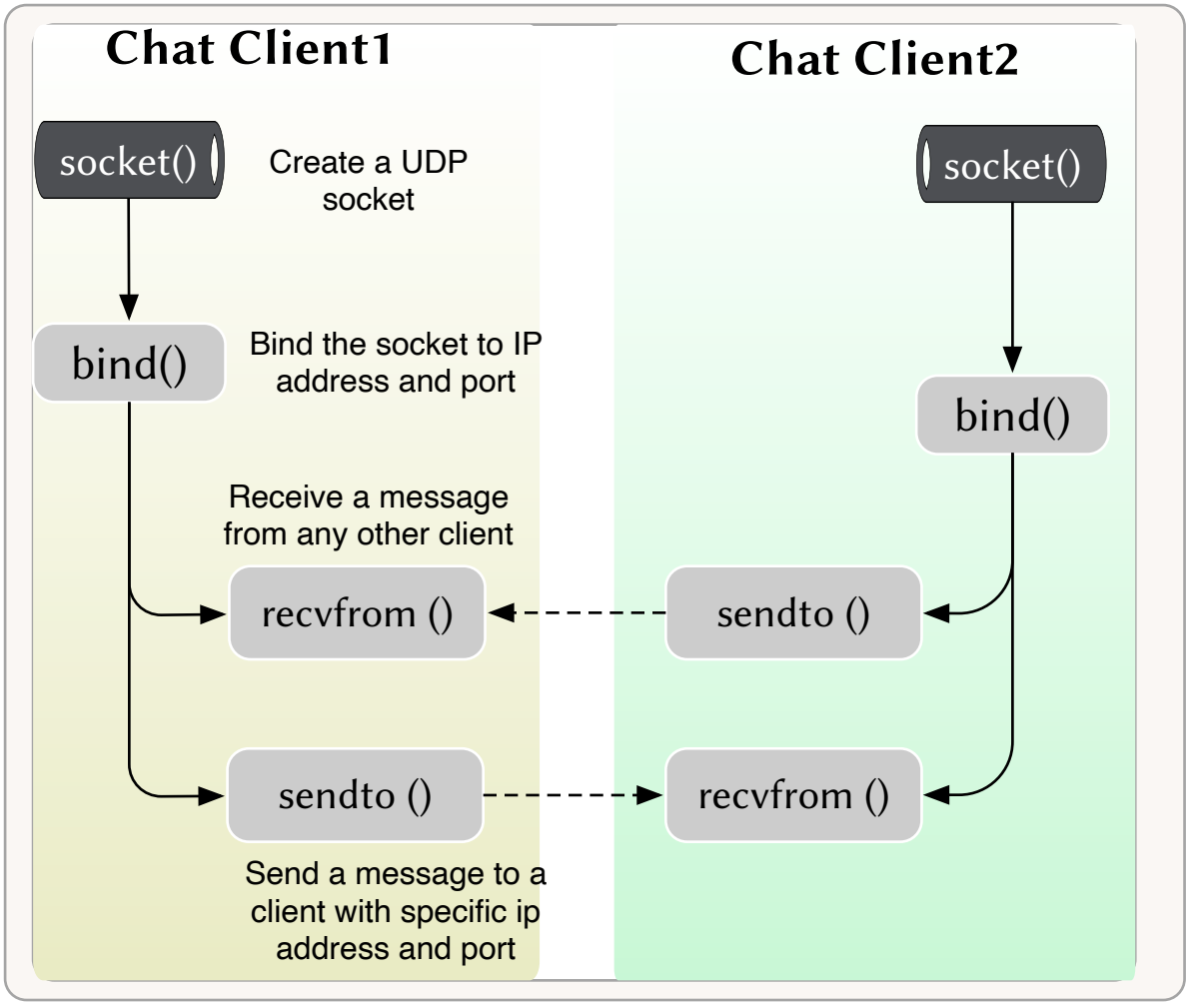

Figure 3: Chat application workflow: Peer-to-peer (p2p) architecture using UDP sockets in python

```python
import socket
from threading import Thread
import sys
import json

# get the IP address of the chatserver (chat.ttm4200.com)
# using your DNS server and socket methods
SERVER_IP = #====fill in here====

# Get the IP address of the client using socket methods
CLIENT_IP =  #====fill in here====

PORT = 5000
RECV_BUFFER = 1024

# Create a TCP socket
TCP_SOCKET = #====fill in here====

# Create a UDP socket
UDP_SOCKET = #====fill in here====

# Make the UDP socket reusable
UDP_SOCKET.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind the UDP socket to the CLIENT_IP and PORT
#====fill in here====

# Dictionary for storing active users
ACTIVE_USERS = None

# Helping function to update active users
def update_active_users(msg):
    global ACTIVE_USERS
    ACTIVE_USERS = json.loads(msg[1:])


# Helping function to get the name of an active user from its IP
def get_name_from_ip_address(ip_address):
    name_list = [name for name, ip in ACTIVE_USERS.items() if ip == ip_address]
    name = name_list[0] if len(name_list) else 'Non-registered'
    return name


# Main thread: connect to the server, then keep listening to receives msgs,
# and send whenever there is an input
def connect_and_listen():
    try:
        # Connect TCP_SOCKET to the chat server
        #====fill in here====

        name = input("First enter your name to register it in the server: ")
        name = name.encode("utf-8")
```

```
            # send the "name" to the server
            #====fill in here====

        except:
            print("could not connect to chat server...")
            sys.exit()

        print(""" Connection Established!
        Now you can enter '#ls' to list active users,
        '@<user1>: msg' send msg to a specific user,
        or just write your msg  to send  to all users
        ***********************************************""")

        # Start a thread to recieve msgs on TCP socket (server connection)
        Thread(target=receive_tcp).start()
        # Start a thread to recieve msgs on udp socket (msgs from other clients, p2p)
        Thread(target=receive_udp).start()
        # Start a thread to send the message when there is input. it will send either
        # to the server (TCP socket) or a specific client using its IP (UDP socket)
        Thread(target=send).start()


    def receive_tcp():
        while True:
            try:
                # Store the message received from the server
                msg = #====fill in here====
                msg = msg.decode("utf-8")

                if msg[0]== '!':
                    update_active_users(msg)
                else:
                    print(f"incoming msg from chatserver ({msg})")

            except OSError:  # Possibly client has left the chat.
                TCP_SOCKET.close()
                sys.exit()


    def receive_udp():
        while True:
            # Receive a message from any other client (p2p chat)
            msg, address = #====fill in here====

            ip_address= address[0]
            sending_client = get_name_from_ip_address(ip_address)
            print (f"Incoming msg (P2P) from {sending_client}:{msg.decode('utf-8')}")


    def send():
        while True:
            # Get input from the user
            usr_input = input(">")
```

```
            # List current active chat users
            if usr_input =='#ls':
                print(ACTIVE_USERS)

            # Send a msg to a single client (p2p) if it starts with "@"
            elif (len(usr_input) > 0) and (usr_input[0] == '@'):
                # Extract the name of the receiving_client and the msg
                temp = usr_input.split(':')
                receiving_client = temp[0][1:].strip()
                msg=':'.join(temp[1:])
                msg = msg.encode("utf-8")

                # Check if the receiving_client is in the active users
                #  (registered int the server)
                if receiving_client in ACTIVE_USERS:
                    # Get the ip_address of the receiving client
                    ip_address =ACTIVE_USERS[receiving_client]
                    # Send "msg" to the receiving client
                    #====fill in here====
                else:
                    print(f"{receiving_client} is not registered")
            # Send a msg to all clients (chatroom),
            # if it doesn't start with a specical character
            elif len(usr_input) > 0 :
                msg=usr_input
                msg = msg.encode("utf-8")
                # Send "msg" to the server (chatroom)
                #====fill in here====


    if __name__ == "__main__":
        connection_threads = Thread(target=connect_and_listen)
        connection_threads.start()  # Starts the infinite loop.
        connection_threads.join()
```

3 Milestone 3 – Running the Chat Applications

- Run the code of the chat server in the “webserver” container.
- Run the code of the chat client in “client1”, “client2” and “client3” containers. Create different names for every client. Send messages to the chatroom and to other clients (P2P).

Q1. **REPORT(5%):** Submit a screenshot of the chat in your report.

Q2. **REPORT(40%):** Submit the completed code in python files (“chatserver.py” and “chatclient.py”).

4 Milestone 4 – Analyzing TCP and UDP Connections

- Start a packet capture with `tcpdump` on your chat server, making sure to dump it to a file (e.g. "chat-server.pcap"), and then start your chat server code.
- Start a packet capture with `tcpdump` on two of your client containers, making sure to dump each capture to files (e.g. "client1.pcap" and "client2.pcap"), and then start your chat client code.
- Send messages to the chatroom and P2P clients and afterwards stop the captures.
- **Save** the capture files in your host machine.
- Open your "chatserver.pcap" in Wireshark and display only TCP packets from and to client1 (`(ip.src==10.20.30.11 || ip.dst== 10.20.30.11) && tcp`). Then answer the following questions and validate your answer with an **annotated** screenshot form Wireshark:

  Q3. **REPORT(5%):** What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the client and the server? What is it, in the segment, that identifies the segment as a SYN segment?

  Q4. **REPORT(8%):** What is the sequence number of the SYNACK segment sent by the server to the client in reply to the SYN? What is the value of the ACKnowledgement field in the SYNACK segment? How did the server determine that value? What is it,in the segment, that identifies the segment as a SYNACK segment?

  Q5. **REPORT(4%):** What is the minimum amount of available buffer space advertised at the receiver for the entire trace? Does the lack of receiver buffer space ever throttle the sender?

  Q6. **REPORT(4%):** Are there any retransmitted segments in the trace file? What did you check for in order to answer this question?

- Open one of your "client.pcap" files in Wireshark and display only UDP packets from and to the other client (`(ip.src==10.20.30.11 || ip.dst== 10.20.30.11) && udp`).Then answer the following questions and validate your answer with an **annotated** screenshot form Wireshark:

  Q7. **REPORT(4%):** Select one UDP packet from your trace (corresponding to chat message). From this packet, determine how many fields there are in the UDP header.

  Q8. **REPORT(4%):** By consulting the displayed information in Wireshark's packet content field for this packet, determine the length (in bytes) of each of the UDP header fields.

  Q9. **REPORT(5%):** The value in the Length field is the length of what? Verify your claim with your captured UDP packet.

  Q10. **REPORT(4%):** What is the maximum number of bytes that can be included in a UDP payload? (Hint: the answer to this question can be determined by your answer to question 8. above) (Note: ignore the overhead from IP or "special" versions of UDP).

Q11. **REPORT(4%):** What is the largest possible source port number?

Q12. **REPORT(3%):** What is the protocol number for UDP? (Hint: this number is displayed in Wireshark as the value of the "protocol:" field in the IPv4 datagram).

Q13. **REPORT(10%):** Submit your capture files ("chatserver.pcap", "client1.pcap", "client2.pcap") along with the report.

5 Optional Exercise

We implemented the chat sever without any authentication mechanism, i.e. the server socket accepts any connection.

Q14. **Extra Credit:** implement an authentication mechanism where only authorized or registered clients can connects to the chat server. Submit your code with a screenshot the authorization process.