Computer Engineering 4DN4
Laboratory 4
Online Grade Retrieval Application

Group 8
Hengbo Huang - 400241747
Yinwen Xu - 400195279

Lab Contribution: Hengbo Huang : server part
Yinwen Xu : client part

Server part:

```
    def get_socket(self):
        #server 的 socket
        try:

            self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)

            ############################################################
            # Bind to an address/port. In multicast, this is viewed as
            # a "filter" that deterimines what packets make it to the
            # UDP app.
            ############################################################
            self.socket.bind(RX_BIND_ADDRESS_PORT)

            ############################################################
            # The multicast_request must contain a bytes object
            # consisting of 8 bytes. The first 4 bytes are the
            # multicast group address. The second 4 bytes are the
            # interface address to be used. An all zeros I/F address
            # means all network interfaces. They must be in network
            # byte order.
            ############################################################
            multicast_group_bytes = socket.inet_aton(MULTICAST_ADDRESS)
            # or
            # multicast_group_int = int(ipaddress.IPv4Address(MULTICAST_ADDRESS))
            # multicast_group_bytes = multicast_group_int.to_bytes(4, byteorder='big')
            # or
            # multicast_group_bytes = ipaddress.IPv4Address(MULTICAST_ADDRESS).packed
            print("Multicast Group: ", MULTICAST_ADDRESS)

            # Set up the interface to be used.
            multicast_iface_bytes = socket.inet_aton(RX_IFACE_ADDRESS)

            # Form the multicast request.
            multicast_request = multicast_group_bytes + multicast_iface_bytes
            print("multicast_request = ", multicast_request)

            # You can use struct.pack to create the request, but it is more complicated, e.g.,
            # 'struct.pack("<4sl", multicast_group_bytes,
            # int.from_bytes(multicast_iface_bytes, byteorder='little'))'
            # or 'struct.pack("<4sl", multicast_group_bytes, socket.INADDR_ANY)'

            # Issue the Multicast IP Add Membership request.
            print("Adding membership (address/interface): ", MULTICAST_ADDRESS,"/", RX_IFACE_ADDRESS)
            self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, multicast_request)
```

Firstly, it will create a socket to receive the messages.

```
def getmes(self):
    self.mes, address_port = self.socket.recvfrom(Server.RECV_SIZE)
    adress, port = address_port
    client =  int(port)
    self.mes = self.mes.decode(Server.MSG_ENCODING)
    if (client in self.client_list ):
        self.connect_CRDS(address_port)
    else:
        self.client_list.append(client)
        self.n = self.n+1
        self.threads(address_port)
```

Then it will decode the message and use the if condition to check whether it is a new client by comparing the client port number saved in the client _list. If not in the list, it means this client is a new client, creating a new thread to serve this client.

```
def connect_CRDS(self,address_port,):
    # self.mes, address_port = self.socket.recvfrom(Server.RECV_SIZE)
    # self.mes = self.mes.decode(Server.MSG_ENCODING)
    # print("Received1: {} {}".format(self.mes, address_port))

    if( self.mes == "connect"):
        text = ("Connected to CRDS, Please enter your command, command list: getdir,makeroom,deleteroom and quit")
        text = text.encode("utf-8")
        self.socket.sendto(text, address_port)
        self.CRDS(address_port)
    elif(self.mes in self.dir_list):
        self.check_room()
    else:
        self.getmes()
```

Once it receives the connect command it will send some tips for clients to use the command. And goto CRDS function, which have Chat Room Directory Server functions.
If it receives the command in self.dir_list it will go to chat_room, which is served for chat between clients. Otherwise, it will go to germes function to wait for the “connect” message.

```
def CRDS(self, address_port):
    self.data, address_portc = self.socket.recvfrom(Server.RECV_SIZE)
    adress, port = address_portc
    client = int(port)
    print("Received2: {} {}".format(self.data.decode('utf-8'), address_portc))
    self.data = self.data.decode('utf-8')
    if (client not in self.client_list ):
        print("create new thread")
        self.client_list.append(client)
        self.n = self.n+1
        self.threads(address_portc)
    else:
        print("1")
        print(address_port)
        adress2, port2 = address_port
        client2 = int(port2)

        if(client2 == client):
            print("2")
            if( self.data == "makeroom"):
                self.makeroom(address_port)
            elif(self.data == "getdir"):
                self.getdir(address_port)
            elif(self.data == "deleteroom"):
                self.deleteroom(address_port)
            elif(self.data == "bye"):
                # text2 = ("Quit CRDS")
                # text2 = text2.encode("utf-8")
                # self.socket.sendto(text2, address_port)
                self.mes = ""
                print("Connection closed")
                self.getmes()

            else:
                text3 = ("command not find, please enter again")
                text3 = text3.encode("utf-8")
                self.socket.sendto(text3, address_port)
                self.CRDS(address_port)
        else:
             self.CRDS(address_port)
```

In this function it will first check whether or not the client is already in the client list, if it is not in the list ,we will create a new thread and append this one to the client list. Otherwise, we will check the received data to decide which command it will use.

```
def makeroom(self,address_port):
    self.mes2 = "Please enter room name"
    self.socket.sendto(self.mes2.encode("utf-8"), address_port)
    self.name_byte, address_portc = self.socket.recvfrom(Server.RECV_SIZE)
    addressc, portc = address_portc
    adress,port = address_port
    if(portc == port):
        self.name = self.name_byte.decode('utf-8')
        print("Received: {} {}".format(self.name, address_port))
        while(self.name == ""):
            self.name_byte, address_port = self.socket.recvfrom(Server.RECV_SIZE)
            address, port = address_port
            self.name = self.name_byte.decode('utf-8')
            print("Received: {} {}".format(self.name, address_port))

        while(self.name in self.dir_list):
            mes7 = "The room name already exists, please change the name of the room"
            self.socket.sendto(mes7.encode("utf-8"), address_port)
            self.name_byte, address_port = self.socket.recvfrom(Server.RECV_SIZE)
            self.name = self.name_byte.decode('utf-8')
            print("Received: {} {}".format(self.name, address_port))
        print ("Room estabilshed\n Back to CRDS, Please enter your command")
        l = self.muticast_address.split('.')
        a= int(l[3])+1
        l[3] = str(a)
        stra = "."
        self.muticast_address = stra.join(l)
        self.port = 2000
        self.get_chatroom_socket()
        self.mes3 = "Room estabilshed\n Back to CRDS, Please enter your command"
        self.socket.sendto(self.mes3.encode("utf-8"), address_port)
        self.CRDS(address_port)
    else:
        self.makeroom(address_port)
```

If makeroom is received, it will go to this function and check whether this room is established. If not it will create a new room which will increment the ip address by 1. If it is an established chatroom , it will send the “already exists” message to the client and inform the clients to change the name. After establishing the room it will go back to the CRDS and wait client’s next command.

```
def get_chatroom_socket(self):
    #給chat room建soket用
    try:

        self.muticast_address_port = (self.muticast_address,self.port)
        self.chat_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.chat_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)

        ##############################################################
        # Bind to an address/port. In multicast, this is viewed as
        # a "filter" that deterimines what packets make it to the
        # UDP app.
        ##############################################################
        self.chat_socket.bind(RX_BIND_ADDRESS_PORT)

        ##############################################################
        # The multicast_request must contain a bytes object
        # consisting of 8 bytes. The first 4 bytes are the
        # multicast group address. The second 4 bytes are the
        # interface address to be used. An all zeros I/F address
        # means all network interfaces. They must be in network
        # byte order.
        ##############################################################
        multicast_group_bytes = socket.inet_aton(self.muticast_address)
        # or
        # multicast_group_int = int(ipaddress.IPv4Address(MULTICAST_ADDRESS))
        # multicast_group_bytes = multicast_group_int.to_bytes(4, byteorder='big')
        # or
        # multicast_group_bytes = ipaddress.IPv4Address(MULTICAST_ADDRESS).packed
        print("Multicast Group: ", self.muticast_address)

        # Set up the interface to be used.
        multicast_iface_bytes = socket.inet_aton(RX_IFACE_ADDRESS)

        # Form the multicast request.
        multicast_request = multicast_group_bytes + multicast_iface_bytes
        print("multicast_request = ", multicast_request)

        # You can use struct.pack to create the request, but it is more complicated, e.g.,
        # 'struct.pack("<4sl", multicast_group_bytes,
        # int.from_bytes(multicast_iface_bytes, byteorder='little'))'
        # or 'struct.pack("<4sl", multicast_group_bytes, socket.INADDR_ANY)'

        # Issue the Multicast IP Add Membership request.
        print("Adding membership (address/interface): ", self.muticast_address,"/", RX_IFACE_ADDRESS)
        self.chat_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, multicast_request)
        self.dir_list[self.name] = [self.muticast_address, self.port,self.chat_socket]
```

This code is for creating a socket for the chatroom. Also, after the chatroom is created , save its name, address, port and socket into dir_list.

```
def getdir(self,address_port):
    print("getdir function")
    print(self.dir_list)
    mes4 = ""
    for i in self.dir_list.keys():
        mes4 = mes4 +i + " "+ str(self.dir_list[i][0])+ " "+str(self.dir_list[i][1]) +"\n"
    self.socket.sendto(mes4.encode("utf-8"), address_port)
    self.CRDS(address_port)
    print("dir sent\n Back to CRDS, Please enter your command")
```

If a getdir command is received, it will go to this function which will send the chatroom information to the client's terminal. The chatroom information is saved in a dictionary variable named dir_list.

```
def deleteroom(self,address_port):
    #这个function
    self.mes5 = "Please enter room name"
    self.socket.sendto(self.mes5.encode("utf-8"), address_port)
    self.dname_byte, address_port = self.socket.recvfrom(Server.RECV_SIZE)
    address, port = address_port
    self.dname = self.dname_byte.decode('utf-8')
    print("Received: {} {}".format(self.dname, address_port))
    self.soket2 = self.dir_list[self.dname][2]
    self.soket2.close()
    del self.dir_list[self.dname]
    mes6 = "Room is delete\n Back to CRDS, Please enter your command"
    self.socket.sendto(mes6.encode("utf-8"), address_port)
    self.CRDS(address_port)
```

If the deleteroom command is received, it will go to this function. First, the client enters the chatroom name that wants to be deleted. Then this function will find the chatroom name in the dir_list, delete it and also close that chatroom's socket.

```
def check_room(self):
    print("into check room")
#     self.input, address_port = self.socket.recvfrom(Server.RECV_SIZE)
#     address, port = address_port
#     if(self.input in self.dir_list ):
#         self.send_messages_forever()

# def send_messages_forever(self):
    try:
        self.socketc = self.dir_list[self.mes][2]
        while True:
            self.chat, address_port = self.socketc.recvfrom(Server.RECV_SIZE)
            self.chata = self.chat.decode('utf-8')
            chat_text = self.chata.split(':')
            if chat_text[1] == 'exit' :
                self.getmes()
            else:
                print("Received: {} {}".format(self.chat, address_port))
                for i in self.client_list:
                    address = '192.168.2.12'
                    address_porta = (address,i)
                    chat_address_port = (self.dir_list[self.mes][0],self.dir_list[self.mes][1])
                    self.socketc.sendto(self.chat,address_porta)

            ##########################################################
            # Send the multicast packet

            # Sleep for a while, then send another.

    except Exception as msg:
        print(msg)
    except KeyboardInterrupt:
        print()
    finally:
        self.socket.close()
        sys.exit(1)
```

Chat_room function is used when a client enters a chatroom, it will send and receive the message to the members who enter in this chatroom.

Client parts:

```
def create_send_socket(self):
    try:
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

        ############################################################
        # Set the TTL for multicast.

        self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, Client.TTL_BYTE)

        ############################################################
        # Bind to the interface that will carry the multicast
        # packets, or you can let the os decide, which is usually
        # ok for a laptop or simple desktop.

        # self.socket.bind((IFACE_ADDRESS, 30000))
        self.socket.bind((IFACE_ADDRESS, 0)) # Have the system pick a port number.

    except Exception as msg:
        print(msg)
        sys.exit(1)
```

Firstly clients will create a send socket which can send the command to server.

```
def __init__(self):
    self.connect_to_server()
    self.send_console_input_forever()

def send_console_input_forever(self):
    while True:
        try:
            print("Command list: connect, name, chat")
            self.command = input("Command: ")
            if self.command != '':
                self.get_cmd()
        except (KeyboardInterrupt, EOFError):
            print()
            print("Closing client socket ...")
            self.socket.close()
            sys.exit(1)

def get_cmd(self):
    if (self.command == "connect"):
        self.socket.sendto(self.command.encode(Server.MSG_ENCODING),MULTICAST_ADDRESS_PORT)
        self.communicate_server()
    elif (self.command == "chat"):
        #self.socket.sendto(self.command.encode(Server.MSG_ENCODING),MULTICAST_ADDRESS_PORT)
        self.chat()
    elif(self.command == "name"):
        self.name()
```

Next it will print some tips to help clients know what command they can use. After the client enters the command, the connect command will be sent to the server for connecting to the

CRDS.

```
def chat(self):
    print("Please enter the chat room name")
    self.room_name = input("chat room name:")
    if(self.room_name in self.dir_list):
        print("Enter the chat room")
        self.socket.sendto(self.room_name.encode(Server.MSG_ENCODING),MULTICAST_ADDRESS_PORT)
        chat_address_port = ((self.dir_list[1]),int(self.dir_list[2]))
        receiver_thread = threading.Thread(target=self.receive_chat_messages)
        receiver_thread.start()
        while True:
            self.input = input()
            self.chat_text = self.chatname+":"+ self.input
            if(self.input == "exit"):
                self.send_console_input_forever()


            # Send string objects over the connection. The string must
            # be encoded into bytes objects first.
            self.socket.sendto(self.chat_text.encode(Server.MSG_ENCODING),chat_address_port)
```

If the command is chat, it will let the client enter the chatroom name and start to chat with other clients who get into this chatroom. It will check whether it is “exit” or not. If the sending message is exited, it will leave the chatroom and return to the command window. Here the message is sent and received in this format: “chatname: message”.

```
def name(self):
    print("Please enter your chat name")
    self.chatname = input("Enter chat name:")
```

Name function is for the client to set a name to use during setting.

Output results:

1. Connect to CRDS:

```
PS C:\Users\22749> cd Desktop/Year4/DN/Lab4
PS C:\Users\22749\Desktop\Year4\DN\Lab4> python lab4.py -r server
Multicast Group:  239.0.0.10
multicast_request =  b'\xef\x00\x00\n\xc0\xa8\x02\x0c'
Adding membership (address/interface):  239.0.0.10 / 192.168.2.12
Connection received from ('192.168.2.12', 57749).
```

When there is a client connected to server, server will print the message prompt that there is client connection.

```
Command list: connect, name, chat
Command: connect
Received:  Connected to CRDS, Please enter your command, command list: getdir,makeroom,deletero
om and quit
Input:
```

For the client part, after entering connect, there will be a message that promptly connects to CRDS, and all the commands that can be used.

2. Makeroom function

```
Received:  Connected to CRDS, Please enter your command, command list: getdir,makeroom,deletero
om and quit
Input:makeroom
Received:  Please enter room name
Input:chat1
Received:  Room estabilshed
 Back to CRDS, Please enter your command
Input:
```

3. Getdir Function

```
 Received:  Room estabilshed
  Back to CRDS, Please enter your command
 Input:getdir
 Received:  chat1 239.0.0.11 2000

 Input:
```

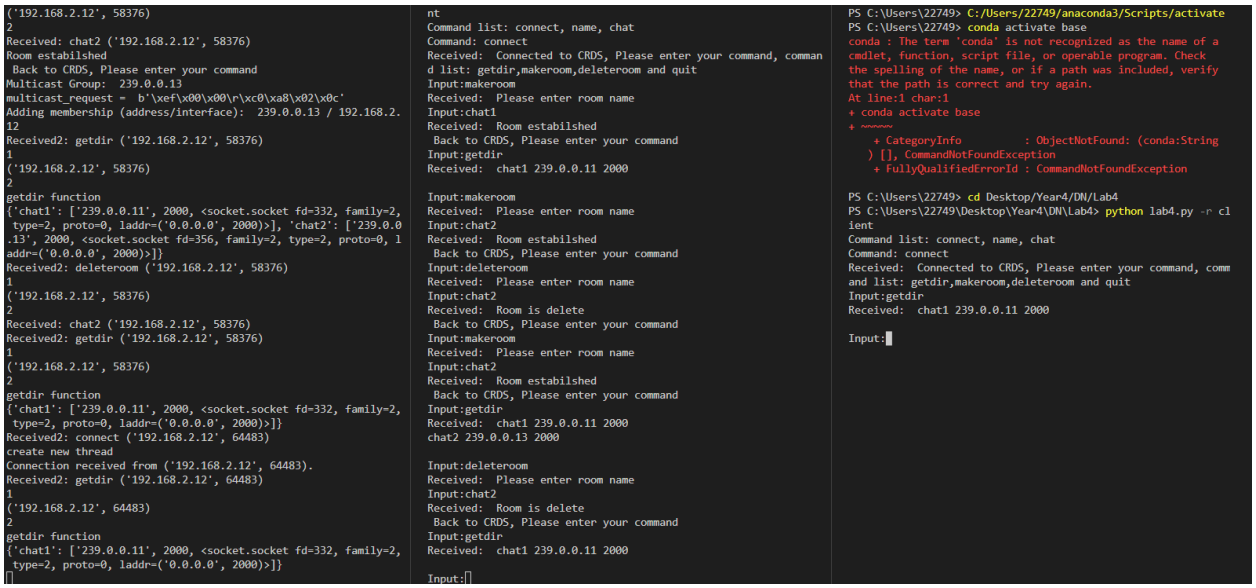
4. Deleteroom Function

```
 Back to CRDS, Please enter your command
Input:makeroom
Received:  Please enter room name
Input:chat2
Received:  Room estabilshed
 Back to CRDS, Please enter your command
Input:getdir
Received:  chat1 239.0.0.11 2000
chat2 239.0.0.13 2000

Input:deleteroom
Received:  Please enter room name
Input:chat2
Received:  Room is delete
 Back to CRDS, Please enter your command
Input:getdir
Received:  chat1 239.0.0.11 2000

Input:
```

5. Bye Function

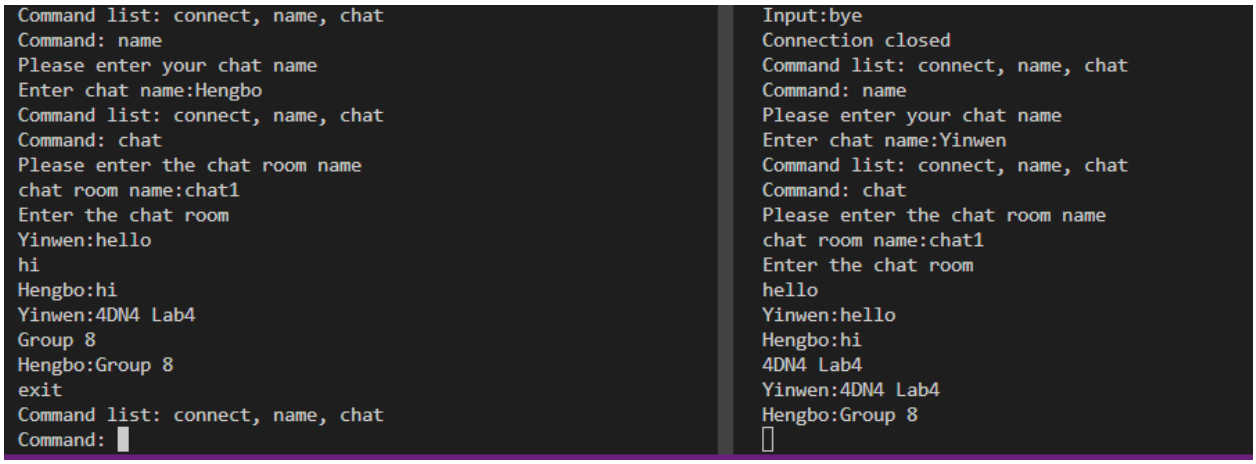
```
Input:bye
Connection closed
Command list: connect, name, chat
Command:
```

6. Server serve for two client



7. Chat Function

Client part:



Server part: