

Problem Class 3

Software Systems

Socket Programming

An application process running on a network needs to open a socket to the transport layer protocol, either UDP or TCP, to begin communication with another application process somewhere else on the network. The purpose of this class is to get some hands-on experience of socket programming, in order to help us better understand the application and transport layer concepts which will be covered in future lectures.

1. Client and server processes on the same computer

UDP sockets are connectionless and do not provide reliable communication. Begin this exercise by simply getting the standard UDP client-server code (shown in class) to work on your computer. This code has been provided in the `udpclient/udpservice` files uploaded with this statement.

The code is written in python. If you do not have it installed already, download python from here and install it: <https://www.python.org/downloads/>

Your installation will include IDLE – an IDE for python. We'll be using the IDLE shell – an interactive Python interpreter – to run application processes on our computer(s).

Important: each IDLE instance will serve as a unique application process. When we wish to run more than one application processes on our computer, we shall run more than one IDLE instances.

Please watch the video **v1**, for instructions on how to run the client and server processes on separate IDLE instances.

Modify the client and server so that the client inputs a text message from the user and sends it to the server which simply displays it:

Client says: *message from the client...*

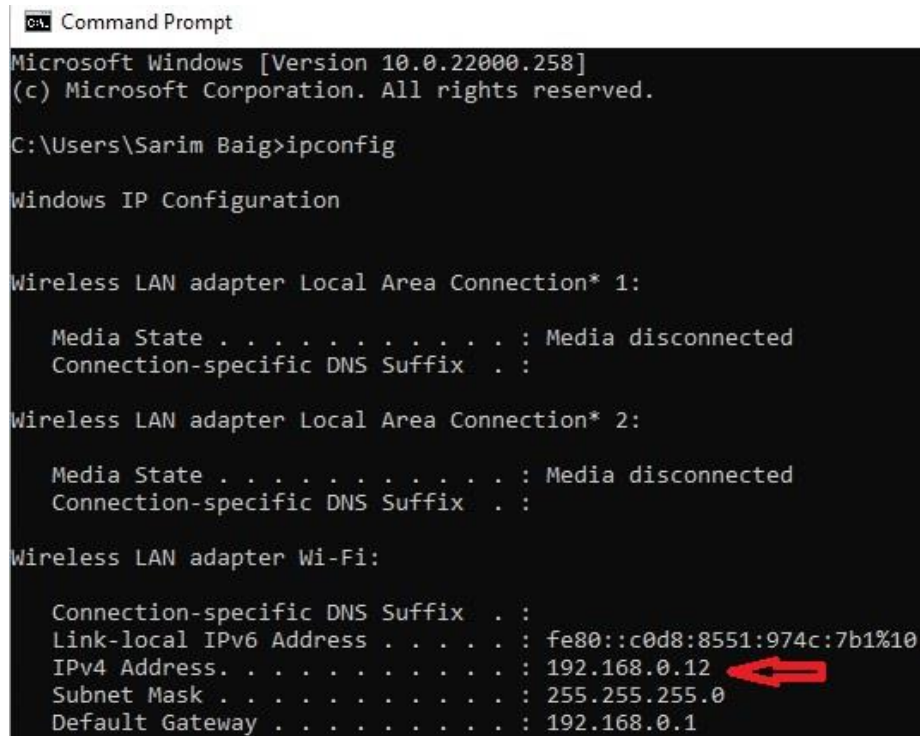
Repeat the same exercise for the TCP client and server.

2. Client and server processes on different computers

Move the client and server processes to two different computers. We will assume that these computers are in the same LAN, e.g., two laptops currently present in your classroom, both connected to the college Wi-Fi.

We need to incorporate the following change in the UDP client:

Instead of sending the message to localhost, the client sends it to the IPv4 address of the server. To see this address, *on the server computer* open command prompt and execute **ipconfig**. You should see the 32-bit IPv4 IP address as show below.



```
Command Prompt
Microsoft Windows [Version 10.0.22000.258]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Sarim Baig>ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection* 1:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 2:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

    Connection-specific DNS Suffix  . :
    Link-local IPv6 Address . . . . . : fe80::c0d8:8551:974c:7b1%10
    IPv4 Address. . . . . : 192.168.0.12
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.0.1
```

Run the client and server IDLE processes as before, but this time on different computers.

Note: localhost is an IP address (127.0.0.1) used by a machine to identify itself. When you change this address to any other address, the machine then looks for the node with that address on the network.

Repeat the same exercise for the TCP client and server.

3. Two clients and a server

In this exercise, we will run two client processes, both of which talk to a single server process.

Create two UDP clients, c1 and c2. Bind c1 to port 11000 and c2 to port 13000.

The server should run on port 12000.

Write code in both c1 and c2 to repeatedly send messages to the server. The messages should be numbers between 1 and 100, sent in sequence, rolling back to 1 after 100. So the clients are infinite loops sending these messages to the server. Add a small amount of delay in the loops so that the processes can be observed easily. You can use **time.sleep(value)** to add delay.

The server should accept messages from c1 and c2, and simply print which message was sent by which client. It should print the messages in the order they were received in.

The server will need to distinguish between the two clients. If all three processes are running on the same computer, they have the same IP numbers, so the port number will be needed to distinguish them. You can access the port number from the `caddr` variable, which is an array containing the IP and port number of the client at `caddr[0]` and `caddr[1]` respectively.

Further Programming Exercises

2-way chat application

These exercises are *optional*, intended for students who may be interested in doing more client server programming in order to understand some of the challenges which client-server architecture can present to an application programmer.

1. We would like to extend our message sending client-server app to a 2-way UDP based text chatting app. This will involve two clients and server.

We will test the application on a single computer, but it can be easily extended to separate computers using their IP address, as explained above.

Watch video **v2** to understand what is required.

Write all your code in files: `client1.py`, `client2.py` and `chatserver.py`.

For the time being, your app is limited to a strict sequence. Client1 sends the first message and then the messages alternate between client 1 and 2.

2. Make your application flexible so that `client1` and `client2` can send messages in any order.
