CSCI 353: Programming Assignment 1

THIS PROJECT MUST BE DEVELOPED IN PYTHON 3.5+

(Incompatible submissions will receive a 0)

Due on September 25, 11:59pm

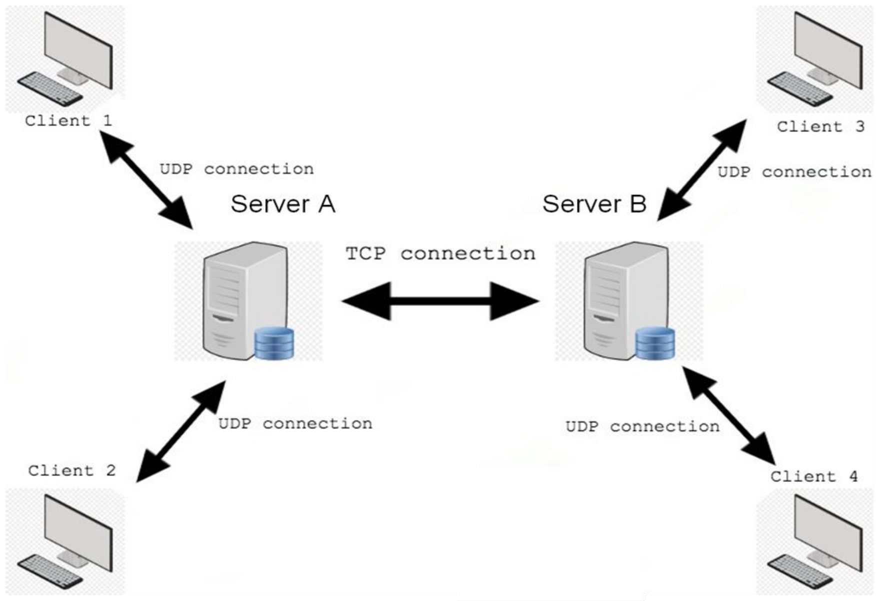
Introduction

The main goal of this assignment is to give you hands-on experience in developing an application using the client-server networking model. In this assignment you will learn how to program with both UDP and TCP sockets and to support higher level application requirements such as transmitting and receiving messages from distributed clients.

This assignment also provides insights into the Internet's best-effort packet forwarding and routing process. This assignment is organized into three parts. While the parts build on each other, the code you submit should work for all the three parts.

You will build a chat system to support messaging between clients. The connections between a server and clients will be UDP based and the connection between servers will be TCP based. For example, the sample topology below shows a network of servers. The clients are connected to the server overlay that allows direct and indirect communication described below:

1. Communication of clients within the same server: If Client1 wants to talk to Client2, it rendezvous through server A as shown: (Client1 → ServerA → Client2)
2. Communication of clients across servers: If Client1 wants to talk to Client3, the message is transmitted to server, which then transmits it to ServerB which forwards it to Client3 as shown: (Client1 → ServerA → ServerB → Client3)



The Client-server Architecture and Connection Protocol Types

Part 1 - One Client and One Server

Develop a chat server and client application using UDP sockets to exchange information across the network. The server runs on a port specified at the command line. The client connects to the server and sends a register message with a client name and then waits for the user input. The client should automatically send register messages to the server after it is started by the user.

Sample UDP code for your reference: https://wiki.python.org/moin/UdpCommunication

Command Line Arguments:

```
> python3 server.py –p <portNumber>
```

where:

-p <portNumber>	port number for the chat server

```
> python3 client.py –s <serverAddress> –p <serverPort> –n <clientName>
```

where:

-s <serverAddress>	chat server IP address
-p <serverPort>	port number of the chat server
-n <clientName>	name of the client (e.g. Tommy Trojan)

Note: If nothing is specified on the command line the server and client programs should print the usage instructions and quit.

The server should terminate upon receiving an interrupt signal (i.e. Ctrl+C). However, the client should terminate either 1) upon receiving an interrupt signal (i.e. Ctrl+C), or 2) if the user types exit in the console. ***Both server and client should terminate without throwing any exceptions.***

Registration Messages:

The messages sent by your application must be in the format described below.

1) The client messages to the server **MUST** be formatted as follows:
   register <clientName>
2) The server messages to the client **MUST** be formatted as follows:
   welcome <clientName>

Sample Execution:

Below is a sample execution for part 1. The following messages should be displayed on the console:

server command

```
> python3 server.py -p <portNumber>
> (Ctrl+C)
```

server output

- ➢ server started on 127.0.0.1 at port <portNumber>
- ➢ <clientName> registered from host <clientIPAddress> port <clientPort>
- ➢ terminating server...

client command

```
> python3 client.py -s <serverAddress> -p< portNumber> -n <clientName>
> exit
```

client output

- ➢ connected to server and registered <clientName>
- ➢ terminating client...

Part 2 – Multiple Clients and One Server

Extend the above server and client programs to support exchange of messages across the network between multiple clients rendezvousing at the server. *You will need to fork a thread for each client connection so that you can handle them concurrently*.

The idea is to have the server handle each client connection in a different thread. When sending messages from the server to clients, you should use the threads you created during client registration instead of generating new threads.

The client and server should support two additional message types, namely **sendto** and **recvfrom**. Additionally, the server should have logs for messages sent to unknown/unregistered clients.

Command line arguments remain the same as part 1.

Communication Messages:

The client messages sent to the server should follow the below convention:

```
sendto <clientName> <message>
```

The server messages sent to the clients should follow the below convention:

```
recvfrom <clientName> <message>
```

You may assume that <clientName> will always be one-word with alphanumerical values only.

Sample Execution:

Below is a sample execution for part 2. The following messages should be displayed on the console.

server command

```
> python3 server.py -p <portNumber>
> (Ctrl+C)
```

server output

- ➢ server started on 127.0.0.1 at port <portNumber>
- ➢ <clientName1> registered from host <client1IPAddress> port <client1Port>
- ➢ <clientName2> registered from host <client2IPAddress> port <client2Port>
- ➢ <clientName1> to <clientName2>: <message1>
- ➢ <clientName1> to <unknownClient>: <message2>
- ➢ <unknownClient> is not registered with server
- ➢ terminating server...

client1 command

```
> python3 client.py -s <serverAddress> -p< portNumber> -n <clientName1>
> sendto <clientName2> <message1>
> sendto <unknownClient> <message2>
> exit
```

client1 output

- ➢ connected to server and registered <clientName1>
- ➢ terminating client...

client2 command

```
> python3 client.py -s <serverAddress> -p< portNumber> -n <clientName2>
> exit
```

client2 output

- ➢ connected to server and registered <clientName1>
- ➢ <clientName1>: <message1>
- ➢ terminating client...

Part 3 - Multiple Clients and Multiple Servers

Extend the server program to support TCP socket connections from one or more servers to create a chat server overlay network. Clients can now chat to other indirectly connected clients on the chat server overlay. Sample code to send and receive data by TCP in Python: https://wiki.python.org/moin/TcpCommunication

A server may have multiple simultaneous TCP overlay connections. When a server receives a sendto message from a client, it should forward the message to the receiving client if it is locally connected or forward the message to its connected server if the client is not local. Each receiving server processes the message similarly, that is, it will forward it to the client if it is connected locally or if the client addressed in the message is not located at the current server, it will forward the message to its connected server. The command line options for the client program remain the same. The extended command line options for the server program are described below.

Command Line Arguments:

- python3 server.py –p <portNumber> -s <overlayServerIP> -t < overlayServerPort> -o <overlayListeningPort>

where:

-p <portNumber>	port number used for UDP client connections. Required.
-s <overlayServerIP>	IP address of the overlay server, for which you want to connect to. Optional.
-t < overlayServerPort>	port number of the overlay server, for which you want to connect to. Optional.
-o <overlayListeningPort>	port number used for TCP connections from other servers. Optional.

-p is the only required argument. Other arguments are all optional. However, without -o, the server will not be listening to TCP connections from other servers. Similarly, without -s and -t arguments, the server will not be connecting to another server via TCP.

Communication Messages:

Messages sent to another server should follow the below convention:

<clientName1> to < clientName2>: <message>

Sample Execution:

Below is a sample execution for part 3. The following messages should be displayed on the console.

Server1 command

```
> python3 server.py -p <portNumber1> -o <overlayListeningPort1>
> (Ctrl+C)
```

Server1 output

- server started on 127.0.0.1 at port <portNumber1>
- server overlay started at port <overlayListeningPort1>
- server overlay connection from host <server2IPAddress> port <server2Port>
- <clientName1> registered from host <client1IPAddress> port <client1Port>
- <clientName1> to <clientName2>: <message1>
- < clientName2> is not registered with server
- Sending message to overlay server: <clientName1> to <clientName2>: <message1>
- Received from overlay server: <clientName2> to <clientName1>: <message2>
- terminating server...

Server2 command

```
> python3 server.py –p <portNumber2> -s 127.0.0.1 -t <overlayListeningPort1>
-o <overlayListeningPort2>
> (Ctrl+C)
```

Server2 output

- server started on 127.0.0.1 at port <portNumber2>
- server overlay started at port <overlayListeningPort2>
- connected to overlay server at 127.0.0.1 port <overlayListeningPort1>
- <clientName2> registered from host <client2IPAddress> port <client2Port>
- Received from overlay server: <clientName1> to <clientName2>: <message1>
- <clientName2> to <clientName1>: <message2>
- < clientName1> is not registered with server
- Sending message to overlay server: <clientName2> to <clientName1>: <message2>
- terminating server...

client1 command

```
> python3 client.py -s 127.0.0.1 -p< portNumber1> -n <clientName1>
> sendto <clientName2> <message1>
> exit
```

client1 output

- connected to server and registered <clientName1>
- <clientName2>: <message2>
- terminating client...

client2 command

```
> python3 client.py -s 127.0.0.1 -p< portNumber2> -n <clientName2>
> sendto <clientName1> <message2>
> exit
```

client2 output

- ➢ connected to server and registered <clientName1>
- ➢ <clientName1>: <message1>
- ➢ terminating client...

Grading Rubric:

We will be using an automated testing script to grade all submissions, so please make sure that your console messages exactly match the expected outputs.

There will be five tests, each worth 20%. ***No partial credit will be given; you either pass a test and get the full 20%, or you fail a test and lose the entire 20%.*** However, the testing script will be posted to Piazza before the due date.

More specifically, part 1 is worth 20% and has one test. Part 2 is worth 40% and has two tests. Part 3 is also worth 40% and has two tests. All tests will be run independently.

Code and Collaboration Policy

You are encouraged to refer to the socket programming tutorials. You can discuss the assignment and coding strategies with your classmates. However, your solution must be coded and written by yourself. Please refer to the plagiarism policy in the course syllabus.

The submissions will be run through code similarity tests. Any flagged submissions will result in a failing score. Keeping your code private is your responsibility.

Submission Instructions

You can develop and test your code on your local machine. However, we encourage you to commit your changes to your assigned GitHub repo as frequently as possible. For grading purposes, we will only look at your most recent version.

You should also create a README, which includes your name, USC ID, and any additional notes you have for the grader.

The following files are required for this assignment:

- client.py
- server.py
- README.txt