Assignment 1: Implementing a Chat Application

Deadline: **Sunday, 20 February 2022 at 11:55 pm**

The goal of this assignment is to introduce you to socket programming through a simple chat application (like messenger). It will allow users to transfer messages and files. **The assignment must be done individually**, and you are required to use Python.

Grade: Overall this assignment will be **7.5% of your grade.**

Assignment Queries: You should post any assignment-related query **ONLY on Piazza**. Do not directly email the course staff.

Plagiarism: The course policy about plagiarism is as follows:

- This assignment should be done individually.
- Copying other students' code, copying from online solutions, or any other sources is strictly prohibited.
- Students must not share actual program code with other students. All students are responsible for protecting their code files from unauthorized reading (e.g., do not share passwords with other students which may give them access to your assignments)
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection. We will run MOSS on your assignment and compare it with both online solutions as well as solutions from previous years.
- Students are strongly advised that any act of plagiarism will be directly reported to the Disciplinary Committee.

Late Submission: You should submit your work on an assignment on LMS before its due time. If you submit your work late for any assignment, we will award you a fraction of the score you would have earned on the assignments had it been turned in on time, according to this sliding scale:

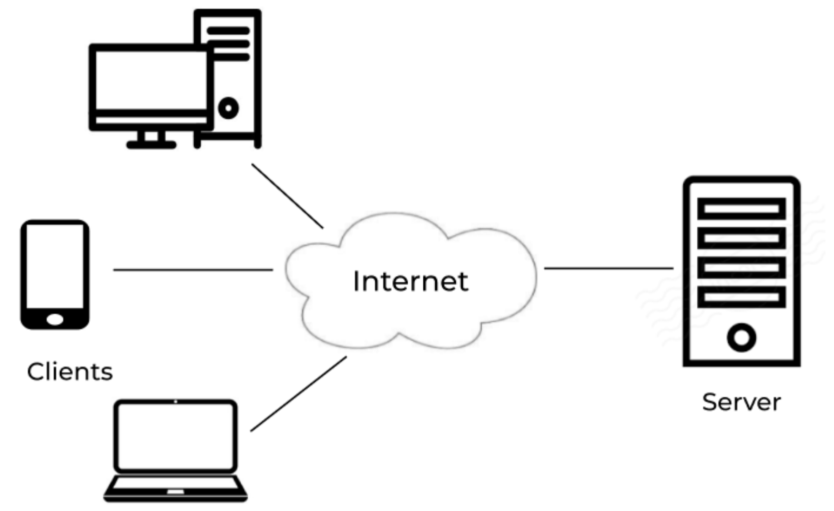
- 90% for work submitted up to 24 hours late
- 80% for work submitted up to 2 days late
- 70% for work submitted up to 3 days late

- 60% for work submitted up to 4 days late
- 50% for work submitted after 5 days late

For example, if you should have earned 8/10 points but submitted 36 hours late, you will instead earn 6.4 points. Besides the above policy, **you are allowed five "free" late days during the semester.** The final assignment will be due tentatively on April 27, 2022. **The last day to do any late submission is also the final day of classes i.e April 29 2022, even if you have free late days remaining.**

Overview

In this assignment, you will implement a chat application (like Messenger). The application architecture is as follows:



There is one server and multiple clients. The clients are the users of the chat application. All clients must implement the functionality needed to exchange messages and files. In this architecture, we will use a central server to keep track of which users are active and how to reach individual users. The server knows all the clients that are currently active (i.e., can receive and send messages) and how to reach them (e.g., active TCP connection). All message and file exchanges happen through the server. A client (e.g., Client_1) that wants to send a message to another client (e.g., Client_2), first sends the message to the server, which then sends it to the destined client.

Your job is to write Server and Client code. You can add your own additional utility functions in util.py file (if you want). Implementation can be done on any platform e.g., Windows, macOS, or Linux. Testing should be done on Linux/macOS machines. Test cases will be available only for programs written on these machines. In case, you have a Windows

machine, we have instructions for setting up either a Linux Virtual Machine (VM) or Windows Subsystem for Linux at the end of this handout.

Simple Chat Application

The main goal of this assignment is to introduce you to socket programming in the context of client-server application architectures. We begin by describing the role of the application server (1.1), application client (1.2), then provide a short introduction to socket programming (1.3), and finally, describe in detail the protocol (1.4) to be used for exchanging messages between the client and server.

1.1) Application-Server

The application server is a multi-threaded server, listening to new connections from clients at a given **host** and **port**, accepting them, and handling the messages sent from each client. The server can concurrently have up to **MAX_NUM_CLIENTS** clients use its service (i.e., while MAX_NUM_CLIENTS are connected to the server, another client can only join if someone disconnects). When a client requests to join a server that already has MAX_NUM_CLIENTS, its request will be rejected. Furthermore, the server does not store information of disconnected clients. The server must handle all possible errors and remain alive until explicitly terminated.

1.2) Application-Client

The application client represents the interface for your chat application, that connects to the application server with a unique username (not already taken by any clients connected to the server). It reads user input from standard input and sends messages to the server, accordingly. It also receives and handles messages from the server. If the client gets any possible error from the server, it will end its connection with the server and shut down. It should show the user an informative message about why it's shutting down.

For the application client and application server to communicate with each other over the Internet, we will use socket programming. Below we describe it.

1.3) Socket Programming

Sockets allow applications to send and receive messages across a network. Sockets are an Operating System mechanism that connects processes to the networking stack. A **port number** identifies a socket. If an application has to send a message, it writes to a specific socket. If an application is expecting to receive messages, it listens on a socket for incoming messages. To create a socket, you need to (i) specify the transport protocol you will use, (ii) the address of the machine (referred to as IP address), and (iii) the port number you will use to identify the socket.

In this assignment, your client should connect with the server using the transport protocol, **TCP.** TCP is a reliable transport protocol that sends a message from the sender to the receiver by first establishing an explicit connection.

Furthermore, you will use the **localhost** as your IP address as you will be running the server and client on the same machine. The port number must be different for each entity as one port can only be used by one process. The server should listen on a fixed port number which all the clients need to know beforehand. The clients should pick a random open port number when creating their sockets. Once you have established a socket, you can send and receive messages.

1.4) Application API and Protocols

Before describing the Application API and protocols in detail, we begin by listing down the sequence of events you should follow for establishing communication between clients.

- First, establish a TCP connection with an already running server and send a "**join**" message from the client to the server.
- The server should add the new client to its list of clients and store its TCP connection details so that it can send messages back to the client.
- Any client can then send messages to any other client(s) connected to the server, using the protocol described later.

For example, the user **client_spiderman** may type:

```
msg 2 client_superman client_batman Hey there folks, Marvel is better than DC!
```

- The client application should interpret this as a message being sent to '2' other clients, namely client_superman and client_batman. And the message text is 'Hey there folks, Marvel is better than DC!'
- The client should then compile a message to send to the server. The server must receive the message, understand that it is incoming from client_spiderman and the 2 intended recipients are client_superman and client_batman.
- The recipients, client_superman, and client_batman, should then receive a message from the server and display it on the screen to their respective users: `msg: client_spiderman: Hey there folks, Marvel is better than DC!`
- This describes the basic crux of the chat application; you must build up from here to include all the required functionality and exception handling, using the protocols described below.

Below we describe in detail the Application API and the protocols you will need to implement.

1.4a) Application API

Each Application Client should be able to perform the tasks given below. For each function, the client must get the user input from standard input (stdin) and process the input according to the below-mentioned formats. If the input does not match with any format, the client should print on stdout:

incorrect user input format

Your chat application should support the following API:

1) Message:

Function: Sends a message from this client to other clients connected to the server using the names specified in the user input. The application server must ensure that the client (whom the message is sent to) will only receive the message **once** even if his/her username appears more than once in the user input.

Input: **msg <number_of_users> <username1> <username2> … <message>**

2) Available Users:

Function: Lists all the usernames of clients connected to the application-server (including itself). **All names must be in one line in ascendingly sorted order.**
Input: **list**

3) File-Sharing:

Function: Sends a file to other clients connected to the server. The other clients should save the file with the same filename (as specified by the sender) prefixed with their username
(e.g.talal_solution.py).

Input: **file <number_of_users> <username1> <username2> … <file_name>**

4) Help:

Function: Prints all the possible user inputs and their format
Input: **help**

5) Quit:

Function: Let the application server know that the client is disconnecting, close the TCP connection with the application server, print the following message to stdout, and shutdown gracefully.
quitting

Input: **quit**

1.4b) Protocols and Message Formats

A message is a basic unit of data transfer between client and server. The different types of messages that can be exchanged between a client and a server are as follows (the message formats mentioned are given below):

1) join

Message Format: Type 1

Sender Action: This message serves as a request to join the chat application. Whenever a new client comes, it will send this message to the server.

Receiver Action: When a server receives this message, 3 things can happen at the server:

- The server has already MAX_NUM_CLIENTS, so it will reply with **ERR_SERVER_FUL**L message and will print

  disconnected: server full
- The username is already taken by another client. In this case, the server will reply with an **ERR_USERNAME_UNAVAILABLE** message and will print:

  disconnected: username not available
- The server allows the user to join the chat. In this case, it will not reply but will print:

  join: <username>

2) request_users_list
Message Format: Type 2

Sender Action: A client sends this message to the server when it reads a **list** as user input.

Receiver Action: The server will reply with RESPONSE_USERS_LIST message and will print:

request_users_list: <username>

3) response_users_list
Message Format: Type 3

Sender Action: The server will send the list of all usernames (including the one that has requested this list) to the client.

Receiver Action: Upon receiving this message, the client will print:

list: <username-1> <username-2> <username-3> … <username-k> where the usernames are ascendingly sorted.

4) send_message
Message Format: Type 4

Sender Action: A client sends this message to the server.

Receiver Action: The server forwards this message to each user whose name is specified in the request. It will also print:

msg: <sender username>

For each username that does not correspond to any client, the server will print:

msg: <sender username> to non-existent user <recv. username>

5) **forward_message**

Message Format: Type 4

Sender Action: The server will forward the messages it receives from clients. It will specify the username of the sender in the message in the List of Usernames field.

Receiver Action: The client, upon receiving this message, will print:

msg: <sender username>: <message>

6) **send_file**

Message Format: Type 4

Sender Action: A client sends this message to the server. In the place of the message, it will place the file (the filename will be placed before the actual file content, separated by space). Receiver Action: The server forwards this file to each user whose name is specified in the request. It will also print:

file: <sender username>

For each username that does not correspond to any client, the server will print:

file: <sender username> to non-existent user <recv. username>

7) **forward_file**

Message Format: Type 4

Sender Action: The server will forward the files it receives from clients. It will specify the username of the sender in the message in the List of Usernames field.

Receiver Action: The client, upon receiving this message, will save the file and print:

file: <sender username>: <filename>

8) **disconnect**

Message Format: Type 1

Sender Action: The client will send this to the server to let it know it's disconnecting Receiver Action: The server upon receiving this, shuts down the connection and removes this user from its list of online users. The server will also print:

disconnected: <username>

9) **err_unknown_message**

Message Format: Type 2

Sender Action: The server will send this message to a client if it receives a message, it does not recognize, from that client. The server will also print:

disconnected: <username> sent unknown command

Receiver Action: The client, upon receiving this message, closes the connection to the server and shuts down with the following message on screen:

disconnected: server received an unknown command

10) err_server_full

Message Format: Type 2

Sender Action: The server will send this message.

Receiver Action: The client, upon receiving this message, closes the connection to the server and shuts down with the following message on screen:

disconnected: server full

11) err_username_unavailable

Message Format: Type 2

Sender Action: The server will send this message.

Receiver Action: The client, upon receiving this message, closes the connection to the server and shuts down with the following message on screen:

disconnected: username not available

Furthermore, there are 4 different message formats for your chat application as given below. Since the messages are sent/received as strings, the different fields in a message will be separated by a single space character (“ “).

Type: 1

Type	Username

Type: 2

Type

Type: 3

Type	List of usernames

Type: 4

Type	List of usernames	Message

List of usernames in Type 3 and 4 will be formatted as:

Num of users	User1	User2	…	UserN

2) Instruction for Setup and Starter Code

The assignment requires you to use a **Linux distribution** and **python3.** Please install a VM (e.g. using VirtualBox) or WSL to set this up. Instructions on setting up VM or WSL are mentioned at the end of the handout.

The starter code consists of:

1. server.py and client.py are the files that you have to implement.
2. util.py contains some constants and utility functions (making messages) that you can use. You can also add your own utility functions to it (if you want).
3. TestChatApp.py is the test file that you can run to check your implementation of client and server code.

3) Manual Testing

To run the server of the chat application, execute the following command:

```
$ python3 server.py -p <port_num>
```

Similarly, execute the following command to run a client (with the same port_num that you have provided to server.py):

```
$ python3 client.py -p <server_port_num> -u <username>
```

To test your submission, we will execute the following commands:

```
$ python3 TestChatApp.py
```

4) Grading

You can earn up to **15 points** from this assignment.

Points Breakdown

1. **SingleClientTest - 3 points**
   A single client can exchange messages with the server.
2. **MultipleClientsTest - 3 points**
   Multiple clients can exchange messages with each other.
3. **FileSharingTest - 3 points**
   Multiple clients can exchange files with each other
4. **ErrorHandlingTest - 3 points**
   Your code can handle both server and client errors.
5. **Manual Grading - 3 points**
   Checking good coding practices such as:
   - Appropriate naming conventions followed. More on this: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names
   - Code modularity
   - Descriptive variable and function names

Notes and Additional Tips

1. First, start early. The assignment is not very difficult but there is a lot of new information that might overwhelm you, so it's important that you give it time. Read this handout carefully. Also, debugging will take a lot of time so make sure you avoid deadline day panic.
2. Unless otherwise stated, each line should end with a newline character. The delimiter between each word of your input and output should be a single space.
3. Google is your friend. Google can give you answers to many questions faster than any TA can. Use these resources effectively. Questions ranging from string manipulation to list trimming or parsing through user input and dictionaries; these can be answered in less than a minute with a few simple keywords being searched for.
4. All the output specified in this document is done to stdout. Since we will be using stdout for testing, you must ensure that only the specified output goes there.
5. You are provided with a helper function in **util.py**, that you are recommended to use.
6. **TestChatApp.py**, invokes instances of client.py and server.py by themselves so you do not need to have them running when testing. However, we recommend that you start working on your code by testing your server.py and client.py files manually since that will be a better way to track your progress in the earlier stages.
7. The test cases are very robust and rigorously check your standard output. **Make sure that you are printing your output exactly as specified in 1.4b and are not printing extra spaces in between or extra newline characters at the end.** These are very easy to overlook and frustrating to debug, so be extra careful.

8. Your server code **must not crash** at any point unless explicitly being told to do so. Make sure you have done exceptional handling to preclude failures.
9. Your code MUST follow pep8 standards.

Setup Guidelines for NetCen Assignments

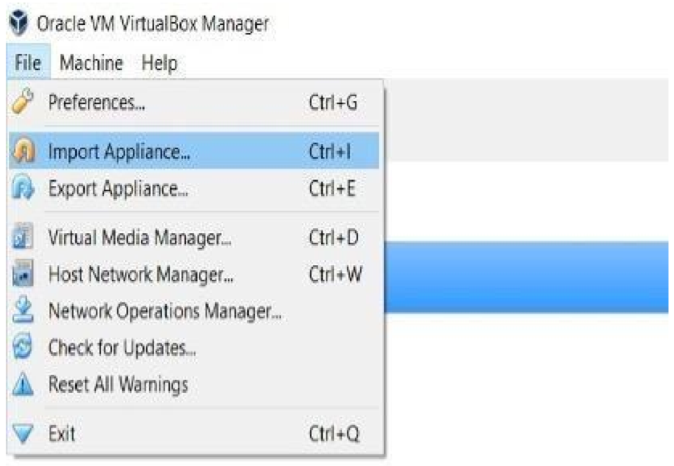
Method 1: Setting up a Linux Virtual Machine Environment

1- Download Virtual Box for your required OS from: https://www.virtualbox.org/wiki/Downloads

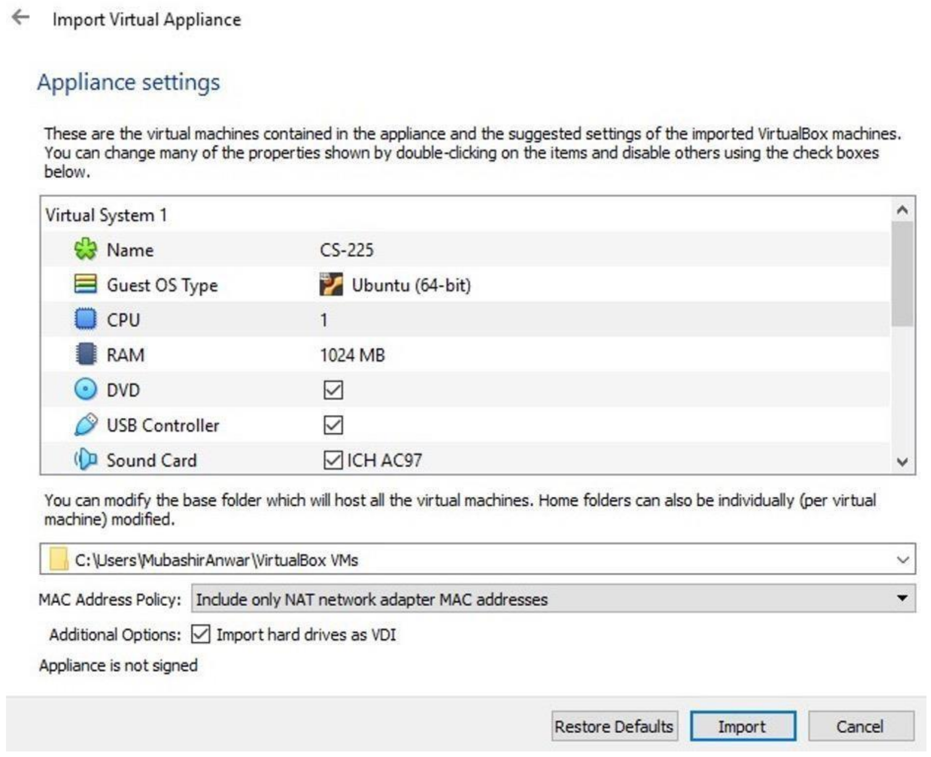
2- Download the .ova file from the given link:

https://drive.google.com/file/d/1c_pIBphjP73VWJsHDDNW_38xz50AmP6E/view?usp=shari ng

3- Assuming you already have your OVA file ready, in VirtualBox, you just have to go to “File -> Import Appliance” in the menu bar.


Oracle VM VirtualBox Manager
File
Machine
Help
Preferences... Ctrl+G
Import Appliance... Ctrl+I
Export Appliance... Ctrl+E
Virtual Media Manager... Ctrl+D
Host Network Manager... Ctrl+W
Network Operations Manager...
Check for Updates...
Reset All Warnings
Exit Ctrl+Q

4- In the dialogue box, add the path to the ova file. The window will then show you the configuration of the current virtual appliance. You can scroll through the configuration list and double click on any item (or check/uncheck the box) to make changes to it. Lastly, click “Import”.



Import Virtual Appliance

Appliance settings

These are the virtual machines contained in the appliance and the suggested settings of the imported VirtualBox machines. You can change many of the properties shown by double-clicking on the items and disable others using the check boxes below.

Virtual System 1	
Name	CS-225
Guest OS Type	Ubuntu (64-bit)
CPU	1
RAM	1024 MB
DVD	☑
USB Controller	☑
Sound Card	☑ ICH AC97

You can modify the base folder which will host all the virtual machines. Home folders can also be individually (per virtual machine) modified.

C:\Users\MubashirAnwar\VirtualBox VMs

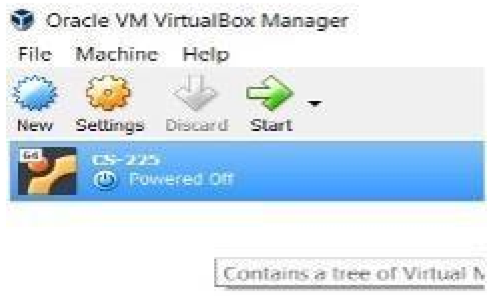
MAC Address Policy: Include only NAT network adapter MAC addresses

Additional Options: ☑ Import hard drives as VDI

Appliance is not signed

5- VirtualBox will proceed to import the virtual machine into your library. Depending on the size of your OVA file, the import process could take quite a while.

Once the process is completed, you should be able to see the new VM in your list.



Oracle VM VirtualBox Manager
File Machine Help
New Settings Discard Start
CS-225
Powered Off
Contains a tree of Virtual M

6- Now you can run an instance of the given OS in the virtual box by clicking on start and proceed to doing your assignments. **The password for the OS is cs225.**

Method 2: Windows Subsystem for Linux (WSL)

On Windows 10, WSL feature allows you to create a linux environment without the need for virtual machine or a separate computer. It allows you to run all the linux commands and services using the command console.

Here's an easy to follow guide with different methods to help you set up WSL on your machine:

https://www.windowscentral.com/install-windows-subsystem-linux-windows-10

If you run into any issues with setting up the Linux environment, you can reach out to your assigned TAs.

Good luck with the assignment! ☺