Introduction to Sockets in Python
CPSC 3600 Networked Systems

This assignment is intended to help you get accustomed to using sockets for network communication in Python. The same concepts that you'll learn here with Python will apply to essentially any other language that also has access to the low-level Socket API.

Note that we will **not** use selectors in this assignment. They will be introduced in a future project.

1 Assignment Instructions

You'll find two Python files in the project template. Each file contains some stubbed out functions that the testing framework will call. I've also included comments outlining the major functionality of each function.

1. **buffered_client.py**: the constructor accepts three values: *a server hostname*, *a server port*, and a *buffer size* to use in calls to recv(). Upon creation, it should create a socket and connect to the server at the specified hostname and port. You also need to implement three functions, described below, that will be called by the autograder.

   (a) ***send_message(message)*** accepts a string message, encodes in the format described below, and then sends it to the connected server via a socket.

   Nothing is returned by this function.

   (b) ***receive_message()*** attempts to receive a message from the connected socket, and upon doing so unpacks it based on the format described below and returns the payload. The server may send a message that exceeds the buffer length, so you must buffer incoming messages until you have received all of a given message.

   This function should return two values: 1) the payload received and 2) a boolean indicating whether or not a message was received. In the event that a message was not received, return an empty string as the payload and a False value.

   (c) ***shutdown()*** cleans up any allocated resources (just the socket in this project).

   Nothing is returned by this function.

2. **buffered_server.py**: the constructor accepts three values: the *a range of IP addresses the server is willing to accept messages from*, the *port to listen to*, and the *buffer size* to use in calls to receive. Upon creation, it should create a socket and bind it to the the specified range of IP addresses and port. You also need to implement two functions, described below, that will be called by the autograder.

   I recommend you implement helper functions for sending and receiving messages like you need to create for the buffered_client.py file.

   (a) ***start()*** begins listening for connection requests. Upon accepting a connection request, the server continously listens for messages from the connected client until the connected client disconnects. Upon receiving a message, the server unpacks it and removes the first 10 characters from the payload (you can use python's slice syntax for this). This truncated message is then sent back to the client(). The server should then begin listening for another message from the client.

   When a client disconnects, the server should begin listening for a new connection request. Upon receiving a new connection, the process described above begins again. The server should continue this behavior until the variable *keep_running* is set to False, at which point the server should end listening for new messages and connection requests and exit the start() function. *keep_running* should be defined as an instance-level variable (i.e. stored in self) in the server's *__init__* function and its value should be set to True. The autograder will set this variable to False when it wants the server to shutdown. Do not set *keep_running* to False in your code.

   The client may send a message that exceeds the buffer length, so you must buffer incoming messages until you have received all of a given message. Start does not return anything upon completing.

   Nothing is returned by this function.

(b) ***shutdown()*** cleans up any allocated resources (the server socket and any other sockets you haven't already closed).
Nothing is returned by this function.

1.1 Message Format

Messages should be packed using the format [payload_length][payload], where [payload_length] is a 4-byte unsigned integer and [payload] is an encoded variable length string. [payload_length] should store the number of bytes in [payload]. Different variables stored within the packed message are represented using square brackets; *the square brackets themselves are not part of the message*. The contents of the square brackets should be encoded to bytes for transmission.

An example of the back-and-forth between the client and the server is shown below (remember that the server is supposed to remove the first 10 characters of any string it receives and then send that back to the client).

1. **Client message:** [30][Four score and seven years ago]
2. **Server response:** [20][and seven years ago]

2 Using Sockets for Network Communication

The details below review information you will need to complete this assignment.

2.1 Setting up and using a client socket

You will always need to import the socket module when using sockets. The files assumes you import the contents of the socket module using the code *from socket import **, rather than *import socket*.

A socket can be created using the socket constructor. You must pass in two parameters when creating a socket: the address type supported and the transport layer protocol you intend to use. We will typically be using the IPv4 address type (indicated using the constant AF_INET) and either the TCP transport layer protocol (indicated using the constant SOCK_STREAM) or the UDP transport layer protocol (indicated using the constant SOCK_DGRAM).

```
from socket import *

tcp_client_socket = socket(AF_INET, SOCK_STREAM)
udp_client_socket = socket(AF_INET, SOCK_DGRAM)

tcp_client_ipv6_socket = socket(AF_INET6, SOCK_STREAM)
udp_client_ipv6_socket = socket(AF_INET6, SOCK_DGRAM)
```

If you are creating a TCP socket, you will need to connect the socket with the remote server. TCP requires this step as it is a stateful protocol, meaning that both the client and server keep track of information related to this specific connection. UDP is stateless and thus does not need to set up an ongoing connection.

Use the connect() function to connect your TCP client to a remote server at a given IP address and port.

```
from socket import *

tcp_client_socket = socket(AF_INET, SOCK_STREAM)
udp_client_socket = socket(AF_INET, SOCK_DGRAM)

# Note the double parentheses when calling connect. The connect function accepts a single
# parameter. This parameter is a **tuple** containing two values, the server's IP address
# and the port to connect to.
tcp_client_socket.connect((REMOTE_TCP_SERVER_IP , REMOTE_TCP_SERVER_PORT))

# UDP sockets don't have any equivalent to the connect() function required by TCP client sockets
```

Your socket is now ready to begin sending and receiving. Due to the stateful/stateless nature of TCP and UDP sockets, they use slightly different functions to send and receive. TCP uses the functions send() and receive(), while UDP uses sendto() and receivefrom().

Despite some differences, both sockets share two common traits:

1. **Message content must be encoded** All data that is sent over a socket must first be encoded into a byte array. We'll primarily use one of two methods to accomplish this:

   (a) **Encoding strings** If all you're doing is sending a string, you can use the functions encode() and decode() to convert between a string representation and a byte array representation. You can also specify what encoding protocol should be used (ASCII, UTF-8, etc). It defaults to UTF-8 if no protocol is provided.

   (b) **Packed byte arrays** Often we want to send information that is more complex than a simple string, or that can be represented using fewer bytes if it is encoded as another datatype. In this case, we'll use the pack() and unpack() functions to convert a set of variables into a byte array and back into a set of distinct variables. We'll look at pack() and unpack() later in this document.

2. **Specifying a buffer length** Both TCP and UDP require setting a buffer size when receiving data. This determines the maximum amount of data that will be returned from the socket in a single *recv()* or *recvfrom()* call. The buffer should be set to a power of 2 less than or equal to 4096. Short messages may fit with this of the available buffer, in which case no extra work is required. However, if the message length exceeds the available buffer length, the message will have to be received as separate chunks and then reassembled. We'll look at how to do this later in this document.

```
from socket import *

tcp_client_socket = socket(AF_INET, SOCK_STREAM)
udp_client_socket = socket(AF_INET, SOCK_DGRAM)

tcp_client_socket.connect ((REMOTE_TCP_SERVER_IP , REMOTE_TCP_SERVER_PORT))

BUFFER_SIZE = 1024

# TCP's send() function takes in the bytearray to be sent
tcp_client_socket.send("Hello world!".encode())
tcp_response_byte_array = tcp_client_socket.recv(BUFFER_SIZE)
tcp_response = tcp_response_byte_array.decode()

# Since UDP is stateless, you must specify what remote server you are sending messages to.
# Similarly, when you receive a message you also get the address information
# (ip address and port) for the server who sent you the message.
udp_client_socket.sendto("Hello world!".encode(), (REMOTE_UDP_SERVER_IP , REMOTE_UDP_SERVER_PORT))
udp_response_byte_array, remote_server_addr_tuple = udp_client_socket.recvfrom(BUFFER_SIZE)
udp_response = tcp_response_byte_array.decode()
```

You need to close your socket once you are done with it, otherwise it will continue to consume system resources once it is no longer needed. You can do so by calling the close() function on the socket.

```
from socket import *

tcp_client_socket = socket(AF_INET, SOCK_STREAM)
udp_client_socket = socket(AF_INET, SOCK_DGRAM)

...

# Close the sockets
tcp_client_socket.close()
udp_client_socket.close()
```

2.2 Setting up and using a server socket

Setting up a server socket uses many of the same functions discussed above. In network terminology, servers are programs that accept unprompted messages from other applications. In order to do so, sockets must bind to a network address (a range of IP addresses and a specific port) that it is willing to accept messages from. The operating system will then direct all traffic received on that port from valid IP addresses to the socket owned by the server application. This is accomplished using the bind() function.

```
from socket import *

TCP_SERVER_PORT = 4567
UDP_SERVER_PORT = 3456

tcp_server_socket = socket(AF_INET, SOCK_STREAM)
udp_server_socket = socket(AF_INET, SOCK_DGRAM)

```

```
# Note again the double parenthesis. Bind accepts a single tuple specifying which client addresses
# the program will accept data from and what port it is listening on. The empty string indicates
# that the server will accept data from an IP address.
tcp_server_socket.bind(('', TCP_SERVER_PORT))
udp_server_socket.bind(('', UDP_SERVER_PORT))
```

The UDP server socket is now completely set up and can begin receiving data, however, the TCP server socket requires some additional configuration. Specifically, the TCP socket needs to begin listening to the port, and then needs to accept a new connection. The listen() and accept() functions are used to do this. When listening, you can specify a backlog value. This sets the number of unaccepted connections the server will keep track of before rejecting new connection attempts. Calling *accept()* completes the connection process and opens up another slot in the backlog. The program you will be developing in this project will only accept a single connection request at a time, which means a backlog could hypothetically develop. Later, we'll look at how to accept many connections at the same time so as to avoid this.

```
from socket import *

TCP_SERVER_PORT = 4567
UDP_SERVER_PORT = 3456

tcp_server_socket = socket(AF_INET, SOCK_STREAM)
udp_server_socket = socket(AF_INET, SOCK_DGRAM)

tcp_server_socket.bind(('', TCP_SERVER_PORT))
udp_server_socket.bind(('', UDP_SERVER_PORT))

# Allow up to 5 clients in the backlog
tcp_server_socket.listen(5)

# Accept the most recent connection request. This creates and returns a new socket
# that will be used to handle all future communication with this client, as well as
# the network address of the client.
new_connection, addr = tcp_server_socket.accept()

# No additional code is required to get the UDP server socket set up
```

You can now being receiving and sending messages through the sockets using the same sending and receiving functions described above for client sockets. You can use *udp_server_socket* directly to send and receive messages, but *tcp_server_socket* is only used to accept new connections. On calling accept(), a new socket is returned, along with the network address of the client. All future communication with this particular client must use the new socket returned here.

The final useful function we'll discuss is setsockopt(). This gives you access to some lesser used settings for a socket. The particular option of interest to us is SO_REUSEADDR. Sockets do not become immediately available for reuse when they are closed (we'll discuss why when we dig into the full details about the TCP protocol). This can cause problems when you are testing your code, especially if you are repeatedly launching and re-launching your project. In this case, you may get an "Address already in use" error. If you encounter this problem, call the following command **before** binding on your socket.

```
# Tell the operating system that you want to make this port immediately available for reuse
# in another program, skipping the typical TIME_WAIT state associated with closing a socket
my_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

2.3 Packing and unpacking data

You will often need to pack arbitrary collections of data into byte arrays when developing networked applications. This allows us to send related chunks of data in a space-efficient representation. Python uses the pack() and unpack() functions to do this, both of which are part of the struct module (which you will need to import). Both of these functions require you to provide a format string that specifies the type and order of the data to be packed (see the previous link for full details). We'll use the following format string as an example to explain how they work: "*!hH?f10s*" There are two parts to this format string:

1. **Byte order specifier** All format strings must begin by specifying the byte order to be used (big-endian or little-endian). As a standard, data sent over the network uses big-endian, which is indicated by placing an exclamation mark at the start of the format string. All format strings used for network data **must** begin with an "!".

2. **Data format string** The remainder of the string specifies what data will be packed and how it should be represented. The example above specifies that the data will contain the following: a short, an unsigned short, a boolean, a float, and a string containing 10 characters. The final packed byte array will consume 19 bytes in total, as shorts are represented using two bytes, booleans one byte, floats four bytes, and each character in strings encoded using UTF-8 use one byte.

   A character in the format string can be prepended with a number to indicate that it is repeated that many times. This can be seen in the format string where we prepend the "s" character with a 10 to indicate that the string is 10 characters long. This can be used for other format characters as well.

The below example demonstrates how these functions can be used.

```
from struct import pack, unpack

format_string = "!?hH?f10s"

packed_data = pack(format_string, 13, 987, True, 0.3871, "1234567890")
unpacked_data = unpack(format_string, packed_data)

# unpacked_data holds a tuple storing the individual values.
# unpacked_data[0] == 13
# unpacked_data[1] == 987
# unpacked_data[2] == True
# unpacked_data[3] == 0.3871
# unpacked_data[4] == "1234567890"
```

A hardcoded format string can be used when sending strings of fixed length, like the one used in the example above. However, if a variable length string needs to be included, the format string will need to be regenerated for each message. The below example packs up a boolean value and a variable length string into a byte array. Note that len() returns the number of characters in the string, so some modification would be needed if the string were encoded into a representation where each character was represented using more than one byte (e.g. UTF-16).

```
from struct import pack, unpack

first_message = "Hello, how are you doing"
second_message = "I'm doing fine, thank you very much for asking"

# Create a new string that prepends the string indicator with the length of the string to be sent
packed_data = pack("!?" + str(len(first_message)) + "s", True, first_message)
unpacked_data = unpack(format_string, packed_data)
#len(unpacked_data[1]) == 24

packed_data = pack("!?" + str(len(second_message)) + "s", False, second_message)
unpacked_data = unpack(format_string, packed_data)
#len(unpacked_data[1]) == 46
```

2.4 Buffering your data

Packets have a maximum theoretical packet size of 64 kB, however, in practice the maximum packet size is 1.5kB, or even as low as 576 bytes. This means that we'll need to take some additional steps if we want to send messages containing more than this much information. These steps differ for UDP and TCP, due to their respective structure. TCP is a *stream-oriented protocol*, while UDP is a *datagram-oriented protocol* (datagram is the fancy term for a UDP packet). Each call to recvfrom() will return all of the contents of exactly one datagram, while each call to recv() may return data from exactly one packet, part of the data from one packet, or data from multiple packets (we'll explain below).

2.4.1 Receiving data with UDP Sockets

As a datagram-oriented protocol, UDP sockets always return exactly the contents of a single datagram when calling recvfrom(). If the next datagram to be received is longer than the specified BUFFER_SIZE value, UDP will throw an OSError [1]. Generally speaking, UDP works best for small messages that will reliably fit within a single datagram. Ideally, keep datagrams under 576 bytes. If you want to use UDP to send larger chuncks of

[1] Specifically, on Windows, it throws the error: *[WinError 10040] A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram into was smaller than the datagram itself.* I presume a similar error is thrown on other operating systems, though the error code will likely differ

data than this, then your client sending the data must chop the data up into individual datagrams which you then reassemble at the receiver. The client does not have this responsibility when sending messages via TCP sockets.

2.4.2 Receiving data from TCP Sockets

As a stream-oriented process, TCP sockets take incoming packets and add the data they contain to a data stream associated with the socket. Calls to recv() then pull data off of that data stream. BUFFER_SIZE specifies the maximum amount of data to pull off of the data stream. If less data than specified is present in the stream, then all the data will be pulled off. If more data than specified is present in the stream, then only the amount specified will be pulled off. As such, each call to recv() may contain less than the entire data of a packet, exactly the data from a packet, or data from multiple packets.

Less than the entire data of a packet will occur if your BUFFER_SIZE is smaller than the size of incoming packets (e.g. BUFFER_SIZE = 32 bytes). Unlike UDP, this will not throw an Exception. Exactly the data from a packet will be returned if the data stream only contains the data from a single packet, and the BUFFER_SIZE equals or exceeds the packet's length. Data from multiple packets will be returned if the data from multiple packets is waiting in the data stream and the BUFFER_SIZE is greater than the size of an individual packet. As such, we need to take some additional measures to make sure that received data is re-assembled correctly. This requires more complexity than is required for UDP, however it enables more data to be transmitted, and it doesn't require the sender to chop large amounts of data up into individual packets. TCP will handle that for the sender automatically.

In order to be able to reassemble the packets, a common approach is to add a fixed-length header to our data that will help the receiver to determine when all the data has been received. The protocol defined in Section 1 is a simple example of a protocol that tells TCP how long a given piece of data is. The length is encoded into a fixed number of bytes at the head of the string. As such, on receiving a new message, the receiver can examine these bytes to determine how much additional data should be fetched.

As the value returned by recv() is simply as an array of bytes, we can pass the fixed-length bytes representing the length to the unpack function, using a truncated format string that only contains the values we want to unpack. In this example, the format string will only contain a single value, as our header only stores the message length. However, in more complex applications, you'll likely be unpacking a fixed length header that stores multiple values.

```
FIXED_HEADER_LENGTH = 4
...
# Since the header is 4 bytes long, we'll want to fetch the next four bytes in the data stream
# when we start to receive a new message
data = new_connection.recv(FIXED_HEADER_LENGTH)

# We can convert the four bytes back into an unsigned integer using unpack. Remember
# that pack always returns a tuple, even if there's only one value in the format string.
# We have to extract the length value from this tuple, hence the [0]
length = unpack('!I', length)[0]
```

Once you have the length of the message, you need to fetch that many additional bytes from the data stream. If the message is short, this could entail a single additional call to receive. If it is longer however, you'll want to make multiple calls to receive and buffer the returned values until you've received enough data. Make sure that you don't ask for more data than you need. This won't cause a problem if all of the data in the data stream belongs to the message you're currently buffering, however, if there is data from additional messages you might wind up fetching the first part of that message, which will corrupt not only the message your currently working with, but also the next message to be read.

```
FIXED_HEADER_LENGTH = 4
...
data = new_connection.recv(FIXED_HEADER_LENGTH)
length = unpack('!I', data)[0]

# Create a new, empty buffer to store the remained of the message
# b"" is Python's syntax for declaring an empty bytes object
payload_buffer = b""

# Check to see if we have received the entirety of the message. If not, receive more data
# and add it to the buffer.
while len(payload_buffer) < length:
    # Be careful not to fetch more data than you need. We check here to see how much data needs
    # to be fetched. If that's more than 1024, we just ask for 1024 bytes of data. However, if
```

```
    # it's less, than we only request the required amount of data
    buffer_size = min(1024, length - len(payload_buffer))
    data = new_connection.recv(buffer_size)
    # recv() returns bytes data. Don't decode it prior to appending it to payload_buffer or you'll
        get an error since payload_buffer is a bytes object.
    payload_buffer += data

# once the while loop has exited, we can decode and return the message
payload = payload_buffer.decode()
```

2.5 Detecting when a connection closes

One of two things might happen if a connection is closed from the other side:

1. A 0-byte array might be returned by the recv() function. This generally occurs when a socket is closed gracefully while a recv() call is pending (i.e. you made the call to recv() prior to the other socket being closed).
2. A "Connection Reset by Peer" exception might be thrown on calling send() or recv(). This generally occurs when making a call to send() or recv() after the other socket has already closed, or if the socket is shutdown abruptly.

Your program should be prepared for both of these eventualities. The first can be detected by always checking that recv() actually returned some data. Python makes this easy to do because a 0-byte array evaluates to False, meaning we can use a simple if... else check.

```
...
data = new_connection.recv(1024)

if data:
    # do something with the data
else:
    # The socket has been closed, handle it however is appropriate for your program
```

Handling a ConnectionReset exception is also simple. You will want to make sure to wrap all calls to send() and recv() within a try... except block, where you catch the exception and handle it appropriately.

```
...

try:
    data = new_connection.recv()
except ConnectionResetError as e:
    # The socket has been closed, handle it however is appropriate for your program
```

3 Test cases

Your program will be graded against three sets of test cases:

1. **Basic client-server communication**
   (a) **Short message** A message shorter than the buffer is sent by the client. The client should receive the same message back from the server.
   (b) **Medium message** A message slightly longer than the buffer is sent by the client. The client should receive the same message back from the server.
   (c) **Long message** A message several times longer than the buffer is sent by the client. The client should receive the same message back from the server.
   (d) **Multiple messages in "series"** The client sends three messages, waiting for a response from the server each time before sending the next. The client should receive each message back distinct from the others.
   (e) **Multiple messages in "parallel"** The client sends three messages and then makes three calls to *recv()*. The client should receive each message back distinct from the others.

2. **Client detection of a server disconnect**

   (a) **Server shuts down** The client sends a short message to the server, which echos it back to the client. The server is then told to begin shutting down. As the server is shutting down, the client begins sending 5 additional messages to the server (we're sending multiple messages as the server won't shutdown immediately as implemented here; it typically shuts down after the first of the five messages is received). On disconnect, the client should detect that the server is shutdown and indicate that the receive was not successful when returning from the *receive_message()* function.

3. **Server response to a client disconnect**

   (a) **Client disconnects, new client connects** The client sends a short message to the server, which echos it back to the client. The client then shuts down, and a new client is created and attempts to connect to the server. The server should detect that the original client has disconnected and begin listening for new clients. It should then accept the new client's connection request. The new client then sends a short message and the server should echo it back.

4 Common Mistakes

1. If you get the error "Your submission timed out. It took longer than 600 seconds to run" when submitting to Gradescope, you are making a blocking call that is never being completed. Based on my interactions with students, this is most likely caused by calling accept() again too soon in your code. See the bold portion in the below instructions about how the server should function:

   > Upon accepting a connection request, the server continuously listens for messages from the connected client until the connected client disconnects. Upon receiving a message, the server unpacks it and removes the first 10 characters from the payload (you can use python's slice syntax for this). This is truncated message is then sent back to the client(). **The server should then begin listening for another message from the client. When a client disconnects, the server should begin listening for a new connection request.**

   Once the server has accepted a connection from a client, it should continue listening for messages from the client until the client disconnects. If you only receive one message and then attempt to accept a new connection, your code will hang and the test cases won't complete.

2. Make sure you import the pack and unpack functions from the struct module at the head of your code
3. Make sure you encode your string prior to packing
4. Make sure that you use the network byte encoding character "!" at the beginning of every format string.

5 Submitting your code

You can submit your project through Gradescope (which can be accessed via Canvas). You'll see the Introduction to Sockets in Python assignment listed on your dashboard. Click on it and a window will appear where you can drag your code files (either directly as files, or as a zipped submission containing the files). A batch of tests will begin running once you submit your code. These should complete fairly quickly, after which you'll see what tests you passed and failed.