Lab: Client server communication using UDP

A simple server and client

You will learn the basics of socket programming for UDP in Python, by starting with a pair of simple server and client programs. The server listens at a specified network interface and port number. The client is given the server network interface as an IP address and also its port number. The client sends to the server a UDP packet containing the current time. The server prints to screen the IP address and port number of the client along with the message received from the client, and sends a reply to the client saying, I received from you a message of this many bytes.

Task 1

You are given the code for the server. Your task is to write the client program. The server program is in the file `udp_server.py`. The line `if __name__ == '__main__'` means this is the main function.

The server program takes two command-line arguments – network interface and port number it listens at. The client program should also take two command-line arguments – IP address and port number of the server. You must run the server program first, by typing:

```
python udp_server.py 127.0.0.1 1234
```

In this first iteration, the server is told to listen at the IP address '127.0.0.1', which is a special one that is reserved for the local interface, meaning that the server can only receive packets locally from clients on the same machine. The choice of the port number 1234 is arbitrary; you should always choose an integer that is 4 or 5 digits in order to avoid colliding with a port number that is already being used by another application.

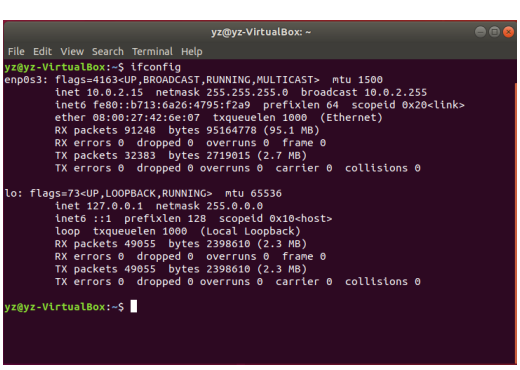
This is an aside for those who have difficulty terminating a running program in command shell. If `Ctrl-C` doesn't work, try `Ctrl-Pause`, i.e., hold down `Ctrl` key while hitting the `Pause/Break` key.

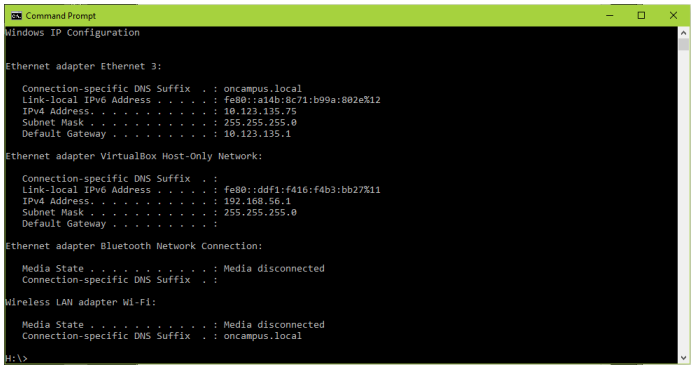
To run the client, type:

```
python udp_client.py 127.0.0.1 1234
```

Try running the client so that it sends the packet to the external network interface of the server, i.e., your machine. First find the external IP address of your machine by typing `ipconfig` in command shell on Windows (`ifconfig` on Linux).

In the figures below, you can see that the Linux machine has the external IP address 10.0.2.15 and the windows machine has IP address 10.123.135.75.





Now run the server as above, listening at the local interface 127.0.0.1, but run the client to communicate with the external IP address of the server (suppose this is 10.0.2.15):

```
python udp_client.py 10.0.2.15 1234
```

The server will not receive the packet sent by the client, because it's not listening at the interface with the external IP address. Now rectify this by running the server:

```
python udp_server.py 10.0.2.15 1234
```

Now if you run the same client again, the packet will get through. You can also tell the server to listen at *all* the interfaces by:

```
python udp_server.py 0.0.0.0 1234
```

In this case, the client can use either the local or external IP address to send packets to the server. In fact, if there is a wireless network interface, the client can also use that.

In line 10 of the server program (udp_server.py), it prints the IP address of the interface and the port number that the server is listening at. You can do the same in the client

program. You will see the client IP address, and the port number that is automatically assigned by the OS to the client.

Note: You are encouraged to write the code on your own for the client. Python documentation might be helpful. Putting your effort into trying will benefit you and help you learn. But if you cannot complete it, you can use the client program in 'udp_client.py' to move forward. **Please refrain from looking at the solution before you have made an effort to do this on your own.**

Unsafe for the client

In line 16 of the client program, 'udp_client.py', a UDP packet is received at the port number assigned to the client by the OS. This is not safe in general, since any program that knows this port number can send data to this client program, without the client knowing that this data is not from the server it was communicating with.

To see this, start the server: `python udp_server.py "" 1234`. Then suspend it by typing `CTRL-Z`. Run the client now: `python udp_client.py 127.0.0.1 1234`. The client will print its assigned port number, **suppose this is 45678, but you should use whatever port number is assigned to your client program**. Open another terminal and start interactive python by typing `python`, and in the Python shell, type:

```
> import socket > sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) >
sock.sendto("I'm fake".encode('ascii'), ("127.0.0.1", 45678))
```

You will see the client receiving this message that is not from the server that the client was expecting. But without further action, the client will not be able to tell that this message is not from the legitimate server.

There are a number of options for dealing with these unwanted illegitimate packets. One thing the client program can add is a check to see if the source address of a received packet is the same as the server address.

Another option is to randomly generate request ID's for packets sent to the server, and having the server put the correct request ID in their reply.

These actions will prevent accidental mistakes but neither will protect against malicious attackers, those can only be dealt with by encrypting the packets.

Remote server and client

Dropped packet or crashed server

We now consider the more realistic scenario in which the server is running on a remote machine that is connected by a network to the machine the client is running on. This means that sometimes packet is delayed or dropped in the network, or the server crashes. Notice that the two situations of a dropped packet and a crashed server are indistinguishable to the client.

In the client program above, line 16 is what's called a blocking call to receive the packet, which means that the client will keep waiting (being idle) until the packet arrives. A dropped packet or crashed server will mean that this client will keep waiting forever.

Timer and retransmit

To avoid this, the client should try to detect if one of those two situations has occurred. One way is to set a timeout on the socket, if the timeout is reached, then it will stop waiting and retransmit the request, hoping a packet was dropped earlier and the server is still up.

There are two issues to consider here. One is what value should be assigned for the timeout, the other is how long should retransmission be tried until giving up altogether.

Suppose a packet was dropped, that is often a result of the network being congested, in which case, it would be unwise to inject even more packets into the network by re-transmitting them. This can be solved by waiting longer and longer to retransmit the packet.

We can double the timeout value every time the previous timer runs out, so the timeout increases exponentially every time a packet is dropped. The client can stop trying to retransmit after the timeout value becomes large enough that it is highly improbable that the server is still alive.

Task 2

To simulate the remote server, a server program called 'udp_server_remote.py' is provided, where packets are selected randomly to be "dropped". Your task is to the write the client program that sets an exponentially growing timer and gives up after timeout value reaches say, 2 seconds. To do this, use the `settimeout(delay)` method of the socket object, and use a `try...except...` on the call to receive packet and catching the `socket.timeout()` exception. (You may need to look up the Python documentation for these methods.) Remember to double the `delay` value every time the socket times out.

Note: Again you are strongly urged to give your best effort to complete your task. However if you truly cannot finish, you can learn from the client program provided, 'udp_client_remote.py'. **Please refrain from looking at the solution before you have made an effort to do this on your own.**

Task 3

For this task, you may wish to pair up with a partner. The two of you will exchange IP addresses of your machines. If you do this on your own, try the packet size suggested first, if that doesn't give you an error, try larger sizes until you get the error.

You will write a simple client that connects to a server at the IP address of your partner's machine, and then simply echo every packet that it receives from the server. So the client and server will be running on two different machines.

You will write the server that tries to send a big packet of 80KB (80000 bytes) to a client; this client process should be running on your partner's machine. The packet can be just one ascii character repeated 80000 times (e.g., `b'z'`). Put this send in a `try...except...else` block. Try to send and catch the `socket.error` exception, if there is one, print something like 'packet too big, cannot send' to screen, otherwise print 'packet sent'. Then you will divide the big packet into 16 1KB packets and send those separately. Observe if this succeeds (it should). Why is this?

To submit

Submit your server program for Task 3. Include your name, student number and lab number in a comment line near the top of your program. Your work for this lab is marked based on effort.