User Datagram Protocol (UDP) :: Worksheet 3

> All work is handed in via git/GitLab and will require the skills demonstrated in this worksheet.

If you have not already done so, complete the setup process for you choice of OS, along with worksheet 1. Instructions for this are on Blackboard under Learning Materials.

For this worksheet you should create a new Git repo and all of your work should be pushed to there. Once you have completed the worksheet and pushed your work to Git you should submit the URL to BB under assignments, worksheet 3.

You repo should contain the following:

- Complete the 3 tasks specified in this worksheet.
- All the Python source code for the worksheet.
- A README.md describing what you have completed, how to run the code, and some examples of each part working.
  - This should be written in Markdown (hence the .md)
- Unit tests that test the functionality of your code.

It is expected for this worksheet that you will need to learn some new things about Python and other technology needed to complete this work. For this you are likely to need to use Google to find out how to peform and use certian libraries in Python and so on.

Marking Scheme

- Each task is worth 25% and will include the following metrics:
  - Correctness of solutions.
  - Comments and overall quality of code.
- Finally, the remaining 25% will include the quality of the READMME.md, the use of Git, and other unspecified additions.

User Datagram Protocol

In networking a useful protocol is the User Datagram Protocol or UDP and is used alongside Transmission Control Protocol (TCP) as core members of the Internet Protocol (IP). In this final worksheet you are going to look in a bit more detail at UDP. Unlike TCP, UDP is a very simply protocol that uses connectionless communication to send out requests to a server and can expect to receive one or more responses.

UDP used a small sized-block, called a checksum, that is derivered from the data to be transmitted, as a simple way to track data integrity. Additional port numbers are utilised for addressing different functions at the server end, necessary there is no explict socket connection from the client to the server.

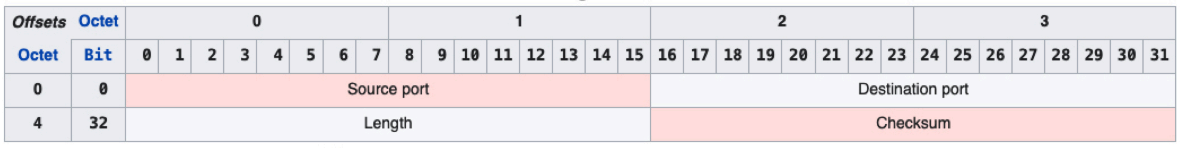
Previously UDP was seen as the less powerful relation to TCP, which is useful for HTTP/2, however, with the development the QUIC protocol and HTTP/3, which is based on UDP, it is has seen a new lease of life. The key observation with QUIC was to note that modern web-pages often have 100 or maybe 1000s of referenced URLs, which are to be loaded concurrently, and the failess nature of UDP at the IP layer, unlike TCP, means that one failing request does not block others.

At the application layer UDP has found numerous successful use cases, such as Open Sound Control (OSC), a lightweight protocol for controlling Digital Musical Instruments over IP.

UDP packet format

A UDP packet is made up of a header and a data seciton (the payload). The former capturing meta information about the message, e.g. the port number, the length of the message and so on, while the later contains the user data being sent in the message.

The header, 8 bytes in size, consists of 4 fields, each 2 bytes, and is defined as:



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

- **Source port** identifiers the sender's port. This field is optional and if not used should be set to 0.
- **Destination port** is the servers port.
- **Length** is the length of the UDP message, including both the heder and the payload. As all UDP messages must contain a header, the minimum length is 8 bytes.
- **Checksum** is used for error-checking of the header and payload. This field is optional or IPv4, but required for IPv6. All packets processed as part of this worksheet will have included checksums.

In practice a UDP packet is contained within the particular IP it is running over, e.g. IPv4 or IPv6, which will contain both the source and destination IP addresses. For this worksheet we will not be concerned with this additional encoding and instead focus solely on UDP.

UDP packets are encoded in Base64. Base64 is a binary-to-text encoding that represents a stream of binary data, split into 8-bit bytes as ASCII strings. Base64 encoding schemes are used when there is a need to encode binary data that needs to be stored and transferred over media that is designed to deal with textual data.

Tasks

All work should be submitted to a Git repo and the URL submitted to Blackboard.

> Work should be demonstrated, and you should include tests to demostrate this, with details in the README how to run them and so on.

Similar to previous tasks there is a server running the remote development machine that transmits and receives UDP packets.

> Note the packets are just UDP there is no IP or transport information included in the packets.

The following URL is used to talk to the server:

```
uri = "ws://localhost:5612"
```

Unlike previous worksheets you will need to not only receive messages from the server, but also decode them as UDP. For example, simply connecting to the UDP server, waiting for a response, will yield the following data:

```
b'CgAqACEAyztXZWxjb21lIHRvIElvVCBVRFAgU2VydmVy'
```

> The prefix b' implies that the following literal is expressed in bytes instead of a string. They may contain non ASCII characters, which values above 127 must be escaped, i.e. a \ must preceed them. As noted above UDP packets are Base 64 encoded and before they can be processed must be decoded.

It is straight forward to decode Base 64 encoded byte streams in Python using the `base64` package:

```
import base64
```

Decoding the above message gives:

```
b'\n\x00*\x00!\x00\xcb;Welcome to IoT UDP Server'
```

The above sequence of bytes are a UDP packet, devided as follows:

- First 8 bytes defined the header:
  - `\n\x00*\x00\x19\x00\xcb\x90`
- The remaining bytes define the payload, which in this case is:
  - `Welcome to IoT UDP Server\x00`

Individual or sequences of bytes can be accessed using Python's slice notation:

- `packet[start:stop]` # items start through stop-1
- `packet[start:]` # items start through the rest of the array
- `packet[:stop]` # items from the beginning through stop-1
- `packet[:]` # a copy of the whole array

where start and stop are offsets into the bytes sequence.

As defined above each field in the header is 2 bytes long:

- Source port identifies the port number of the source and is defined as bytes 0 and 1.
- Destination Port identifies the port of the destined packet and is defined as bytes 2 and 3.
- Length is the length of the UDP packet, including the payload, and is defined as bytes 4 and 5.
- Checksum is the 16-bit one's complement of the one's complement sum of the UDP header, and is defined as bytes 6 and 7.

To read a field from the header you need to access the bytes for that field and convert to an integer value, specifing the endianness. For example, the following reads the length field from the byte stream `packet`:

```
length = int.from_bytes(packet[4:6], 'little')
```

Which in the case of the above packet is 25.

The remaining header fields can be decoded similarly to give:

```
Source Port: 10
Dest Port: 42
Data Length: 25
Checksum: 37067
```

The payload itself can be any kind of binary data and is application dependent. In this case we can safely assume it is a string encoded as UTF-8, and thus we just need to convert it:

```
payload = packet=[8:(size+8)].decode("utf-8")
```

At this point you should have a reasonable idea how a UDP packet can be decoded, putting aside calulation of the checksum.

Task 1

For the first task you need to implement a basic client to receive the hello message from the UDP server and decode it, without error-checking.

Your client should output the received packet in the form:

```
Base64: b'CgAqACEAyztXZWxjb21lIHRvIElvVCBVRFAgU2VydmVy'
Server Sent: b'\n\x00*\x00!\x00\xcb;Welcome to IoT UDP Server'
Decoded Packet:
Source Port: 10
Dest Port: 42
Data Length: 33
Checksum: 15307
Payload: Welcome to IoT UDP Server
```

Task 2

You should now be able to connect to the UDP server and receive and decode the hello message, however until now you have not checked that the packet is valid. To check that the packet is valid a checksum must be computed for the received packet and compared against the checksum supplied in the packets header.

Of course, it would be impossble to create teh checksum for a UDP packet knowing the checksum before hand so it can be included in the header. To get around this the checksum is computed with the complete packet, except that the checksum field is set to 0. The checksum is then computed and inserted in to the packet for transmission.

As noted the UDP checksum is in reality checked at the IP level, as it is optional for IPv4, but required for IPv6. However, for this worksheet we are going to compute it at the UDP level.

The checksum is straightforward to compute as it is simply the one's complement of the UDP packet, with the checksum set to 0. One's complement is simply the value we get when we switch all 0s to

1s, and all 1s to 0s, i.e it is the bitwise negation of the value. For example, consder the 8-bit binary number:

```
1011
```

it's one's complement is then:

```
0100
```

So how does this apply to calculating the one's complement for a UDP packet? To see how this works let's consider a simply example:

```
dest_port = 1
source_port = 2
length = 9
payload = 'AB'
```

In binary the values are:

```
dest_port = 0000000000000001
source_port = 0000000000000010
length = 0000000000001001
payload = [01000001, 01000010]
```

> Note that the payload is simply a sequence of 8-bit binary values, while each of the header fields are 16-bit, i.e. 2-byte, values.

To calculate the checksum, first add together each of the binary values:

```
0000000000000001 + 0000000000000010 + 0000000000001001 + 0100000101000010
```

It is important to note here that the payload values are combined into 2 byte values, with the first value to be included being shifted up 8 bits to form the top 8 bits of the 16 bit value. In the case that there is an odd number of bytes in the payload, a zero is inserted into the final lower byte.

Performing the addition above we get the decimal value 16,718 decimal, which in in binary is:

```
  0000000000000001 + 0000000000000010 + 0000000000001001 + 0100000101000010
=
  0100000101001110
```

Finally, to compute the checksum we simply flip all 0s to 1s, and all 1s to 0s, i.e. perform a bitwise NOT:

```
  NOT(0100000101001110)
=
  1011111010110001
```

If we now add the orignal value, computed from the addition, with the outputed checksum the result will be all 16-bits being set:

```
  0100000101001110
+
  1011111010110001
=
  1111111111111111
```

It is easy to see that if just one bit had got changed, i.e. corrupted, during transfer, the result will be different and thus an invalid packet had been received.

For this task you need to implement the following function, that given a source port, destination port, and a payload computes the corresponding message's checksum.

```
compute_checksum(source_port: int, dest_port: int, payload: bytearray) ->
 ↪  int
```

Task 3

For the final task you need to send a receive UDP packets from the server. If you send the payload:

```
'1111'
```

it will return the Coordinated Universal Time (UTC). For this task you need to modify your server so that it implements something like the following loop

```
async with websockets.connect(uri) as websocket:

        await recv_and_decode_packet(websocket)

        while True:

            await send_packet(websocket, 0, 542, b'1111')

            await recv_and_decode_packet(websocket)

            time.sleep(1)
```

Here the client connects to the server, receives and decodes the welcome to server message, and then enters an infinite loop, sending a valued UDP packet to the server, with the payload `1111`, and waiting to receive the time back, and then sleeping for 1 sec and repeating.

An example output that is display when a packet is received should look something like:

```
Base64: b'HgIAABAAIvwwODozODozNQ=='
Server Sent: b'\x1e\x02\x00\x00\x10\x00"\xfc08:38:35'
Decoded Packet:
Source Port: 542
Dest Port: 0
Data Length: 16
Checksum: 64546
Payload: 08:38:35
```

The payload is UTC time, which in this case would have been 09:38:35 BST.

You will need to ensure that the correct checksum is calulated for the message sent to the server.

> Hint: for this task you will need to convert each header field and the payload into bytes, to acheive this you can use Pyhton's `to_bytes` method. You will likely need to use Google to understand how this works. Remember to use a byteordering of little endian.