Lab 2

Purpose:

- To write a basic client/server application
- To work with raw byte data
- To learn Python's basic socket library

Handing In:

You must submit the following files to the appropriate folder in the D2L Dropbox page:

- `server.py`
- `client.py`

Specification:

You are to write a client/server application in Python using UDP. The client requests a math operation to be applied repeatedly to a sequence of numbers; the server computes and returns the result.

The Client:

The client takes a server address, a port number, an operator ('+', '−' or '*') and up to 10 small integers on the command-line. The integers must be between 0 and 15, inclusive. (You don't need error-checking to ensure there are at most 10 integers, nor that they are in the correct range. I will never pass invalid input to your client.) The client will create a datagram packet that contains this information and send it to the server. The server will return the result of performing the operator on all the integers from left to right.

For example, to run the client so that it connects to a server on the same computer, listening on port 12345, you could do the following:

```
python3 client.py localhost 12345 - 13 5 2 4
```

The server would receive this request and return the result: 2. The client should print out this result plus a newline character with *no* extra output whatsoever.

The datagram packet sent to the server must be built as follows:

- the first byte will represent the operator:
    - a 1 in bit position 0 (the least significant bit) will represent '+'
    - a 1 in bit position 1 will represent '-'
    - a 1 in bit position 2 will represent '*'
    - all other bits will be zero

- the second byte will represent a count of integer values that follow.

- the third and subsequent bytes will hold the integer values to which the operator will be applied. Each byte will hold two 4-bit unsigned integers; the last byte, if there is only one 4-bit unsigned integer remaining, will put the integer in the MOST SIGNIFICANT bits, and pad the LEAST SIGNIFICANT bits with all zeros.

After sending the packet to the server, the client waits and receives the server's result, and displays the calculated number (plus a newline) and quits.

NOTES:

- The command arguments: "- 1 2 3 4 5" actually mean: 1 - 2 - 3 - 4 - 5, so the correct answer is -13. It is natural to assume the "1" would be negative as well, but that is *not* the case.

- There is no "count" on the command-line args -- that is *just* for the data bytearray that is sent to the server. If there are five numbers to add, for example, the bytearray sent to the server would have 00000101 (5) as the second byte, but that number does not appear in the user's command-line argument; it would still be: "+ 1 2 3 4 5", not "5 + 1 2 3 4 5", or something like that!

The Server:

The server essentially runs forever; it enters a loop waiting for a socket connection from the client. When a client connects, it parses the data, performs the requested operations, sends the result back to the client, and returns to its wait-loop. The server can only be stopped by an operating system interrupt (e.g. `CTRL-C` if it is running in a foreground shell process, or with `SIGHUP` -- as in the Linux "`kill <pid>`" command; note on Windows sometimes you need to use `CTRL-PAUSE/BREAK` instead of `CTRL-C`).

The server's return datagram must be exactly 4 bytes in length, with the return result represented as a 32-bit signed integer contained in the 4 bytes. The first byte must be the MOST SIGNIFICANT byte of the return result, and the fourth byte must be the LEAST SIGNIFICANT byte of the return result. Note that most machines we work with are

"little-endian", so this can affect the way you extract the individual bytes that you are returning. Sending the bytes as described here -- with data[0] being the most-significant byte and data[3] being the least-significant byte -- is the typical "network order", which is big-endian.

To start the server you must supply the port number as a command-line argument, as in:

```
python server.py 12345
```

You don't need any specific code that will stop the server; just enter an infinite loop, and the user will kill the server as specified above.

Testing:

Note that client/server applications are interesting, because in theory they can be developed completely independently, by different people, so long as they both share the same specifications!

I can therefore test your code by first running your client and having it send its request to my server; and conversely I can have my client send its request to your server! This way I can be confident you are creating the datagrams as I've specified, and not just sending the data in your own way that both your client and server agree upon!

Test accordingly!

(Hint: you might want to find a partner who has also completed the lab -- preferably someone you didn't work closely with! -- and test your client against their server and vice-versa!)

Using bytearrays:

Both the client and the server will have to work with raw byte data. While the Python library has excellent support for text data, with datagrams, programmers often need to work at a lower level, optimizing space requirements to send the smallest packets possible. We will therefore use Python's `bytearrays` to create our data packets.

A `bytearray` is a built-in data type that is, as you might guess, an array of bytes! It is not a real array, but behaves mostly like one so that you can add and access individual byte data. You create a `bytearray` by using the built-in `bytearray()` function:

```
buffer = bytearray()
```

Optionally you can create a `bytearray` with a specific capacity, and all bytes will be initialized to zero:

```
# creates a 15-byte raw bytearray.
# all elements (0-14) are set to zero
buffer = bytearray(15)
```

Note that `bytearrays` are backed by Python Lists, and therefore can grow as needed without any extra work by the programmer.

To add bytes to a `bytearray`, you just use the `append()` method:

```
buffer.append(13) # Add 00001101 to the byte array
```

You have to add bytes using integers, and the integers must be between 0 and 255 inclusive, or else Python will crash.

To access bytes within the `bytearray`, you just use array notation:

```
value = buffer[3] # retrieve the 4th byte from the bytearray
```

Note that there is a very significant difference between Python 2 and Python 3 with regard to socket programming. In Python 2 it was allowed to send and receive text strings or bytearrays, whichever was more convenient for the programmer. Since Python 3, *only* bytearrays are allowed. This is because it forces the programmer to be explicit about the string encoding, making the resulting code more robust and less dependent upon default behaviour, which can vary from system to system.

Bitwise Operations:

One of the ways using `bytearrays` can be more space-efficient is that you can chop up an individual byte to represent more than a single unit of data. For example, you could use a single byte to represent two 4-bit integers; or you could use a single byte as 8 separate boolean flags (`0 == false`, `1 == true`).

In order to look at parts of a single byte we need to use "bitwise operators". In Python there are 6 bitwise operators:

- `x >> n` - shift the bits from `x` to right `n` times
- `x << n` - shift the bits from `x` to the left `n` times
- `x & y` - do a bitwise "and" between each corresponding position in `x` and `y`
- `x | y` - do a bitwise "or" between each corresponding position in `x` and `y`
- `x ^ y` - do a bitwise "xor" between each corresponding position in `x` and `y`
- `~ x` - flip all the bits in `x`

These operators allow us to do a variety of manipulations on bytes:

- extract the value (zero or one) of the bit at position `N` from byte `x`:

```
mask = 2**N
value = x & mask
```

- extract the value of the first (least significant) `N` bits as an integer:

```
mask = 2**N - 1
value = x & mask
```

- extract the value of the last (most significant) `N` bits as an integer:

```
mask = (2**N - 1) << (8-N)
value = (x & mask) >> (8-N)
```

(We'll go over these calculations and how they work in class and lab; you will be responsible for understanding how to create and use masks like this!)

Suggested Implementation Steps

It is highly recommended that you work on pieces of this lab one bit at a time, testing along the way. If you try to write everything at once, and it doesn't just work the first time (it won't!!), you won't have any idea where to start to fix it!

Breaking up a larger task into smaller, manageable and *testable* steps is a critical skill of any developer!! You can of course decide on your own steps as you like; whatever works for you is perfectly acceptable. But if you'd like some help getting started, consider doing the lab in these steps:

1. Get the basic client/server communication set up. Have your client send your name to the server, and have your server reply with "`Hello, <your name>`". Print out the server's response in your client.

2. Change the message to use a `bytearray` instead of a string. Send a 1-byte integer (as a `bytearray` with one element) to the server, and have the server respond with that value plus one. Display the server's response.

3. Add command-line support to your client. Pass the 1-byte integer as a command-line argument, rather than having it hard-coded. Does your server still respond properly? What happens when you pass in the value of 255?

4. Add support for different port numbers. Have the client and server both take the port number as their first command-line arguments. What happens when the port numbers don't match? What happens when you use a known port number like 80 (`HTTP`)? How about 22 (`SSH`)?

5. Turn the server into a simple adder. Pass two 1-byte integers to the client; have the client turn these into a two-element `bytearray` and send them to the server. The server then retrieves these values, adds them, and returns the sum in a single 1-byte result. What happens if you try to add 150 and 200?

6. Enhance the server to return 2 bytes (because the sum of two 1-byte numbers could take a second byte), thus eliminating overflow. Modify your client to receive the 2-byte answer and display it as a single integer.

7. Add support for the different operators. Have the client take in the operator as the second argument (remember the port number is the first), followed by the two numbers. Pass the operator (using the bit codes specified in the lab) as the first byte, and the two values as the second and third bytes. Modify the server to check the operator code and

apply the correct operation to the values. It still returns the result in as a 2-byte answer.

8. Add the ability to pass multiple 1-byte values to the server. The first byte will still be the operator bit-code; the second byte will be the number of values that will be used; and the third and subsequent bytes will be the values passed on the command-line.

9. Change the code to use 4-bit values instead of 8-bit values. This requires packing two 4-bit values into a single byte before appending to the `bytearray`. Take care to handle the case where you've just got an odd number of 4-bit values; the last of these needs to be packed into the most-significant bits of the last element of the `bytearray`.

Remember to test thoroughly at each step!! And don't read ahead … you can confuse yourself by trying to jump ahead or do multiple steps at once. Each of these steps can be done relatively quickly (compared to the whole thing!), which makes it easier to work on steps throughout the weeks, rather than *just* trying to blast through many steps during a single lab period! It's important to spread out your work, and having small, bite-sized tasks (sorry for the pun!!) makes it easier to schedule a little bit of work in between everything else you have to do!