Course work definition

In this course work you will generate a client program which can receive and send messages according the protocol specified. The program first performs a handshake with the server using TCP and then uses UDP for sending and receiving messages. You will only create a client, not the server, the server will be made available by TAs. Like previously stated the suggested way of creating your program is to use python.

Basic outline of the program:

- Program takes server address and port as command line argument
- Program uses TCP to connect to the server, negotiates possible extra features and encryption keys with server and receives an identity token and the server's UDP port
- Program initiates UDP messaging by sending the first message
- Server sends a message with random words
- Program replies with the words in reverse order
- Server sends new messages random number of times
- Finally, the server sends a message telling the exchange is finished

You must test your implementation against the server created by the TAs. The server is running on the host **195.148.20.105**, TCP port **10000**.

TCP protocol

TCP is used to receive an identification token and the server's UDP port and to negotiate supported features with the server.

- Client connects to the server address and port given as command line arguments
- Client sends a HELLO message to server along with group number, optional features and encryption keys
- Server replies with the client's identification token, the server's UDP port and optionally encryption keys
- Each message must end in carriage return and newline (\r\n)

Example:

Client sends	Server replies
"HELLO\r\n"	"HELLO asd123 12345\r\n"
"Hello from asd123\n"	"one two green seven"
"seven green two one"	"uno dos tres"
"tres dos uno"	You replied to 2 messages. Bye

Negotiating extra features

When implementing extra features, capabilities of the client must be included after the HELLO, separated by spaces.

The following words describe the features:

ENC	Encryption
MUL	Multiple messages
PAR	Parity

Example

All features **(note the last line after the key exchange containing only ".\r\n", this is needed for the encryption to work (see documentation for encryption below))**:

Client	Server
"HELLO ENC MUL PAR\r\n <Key 1>\r\n … <Key 20<\r\n .\r\n	"Hello dfs445 8282\r\n <Key 1>\r\n … <Key 20<\r\n .\r\n

Just multipart:

Client	Server
"HELLO MUL\r\n"	"Hello dfs445 8282\r\n"

UDP protocol

After setting up features and ports with TCP, we use UDP for actual data communication. Structure of the UDP message is following:

UDP packet structure

CID	ACK	EOM	Data remaining	Content length	Content
Char[8]	Bool	Bool	Unsigned short	Unsigned short	Char[128]

- CID: 8 byte string containing the client's identification token received from the server's hello message. May be ignored when receiving from the server.
- ACK: Boolean value whether the last message received correctly or not. Used with the parity check feature, otherwise always True.
- EOM: Set as True in the last message the server will send, otherwise always False
- Data remaining: Length of the data remaining when using multipart messages. If multipart messages are not supported, value is always 0.
- Content length: Length of the message (without padding) in the content field **before encoding**. Use this value to extract the exact message from the content field.

- Content: 128 byte string of message content. If the content is smaller than 128 bytes it will be padded with null bytes. Content length should be used to get only the message. In implementations where there is no parity server sends messages that are at most 64 bytes long and expects messages that are the same size. If you are implementing multipart messages send your data in chunks of 64
- Byte order is network byte order (big-endian)

Messages are packed as binary data. This can be done using struct.pack and struct.unpack, for example:

```
Data = struct.pack('!8s?5s', string1, bool1, string2)
```

This would pack string1 as string with a length of 8, bool1 as a boolean and string2 as string with the length of 5.

If you want to unpack the above packet use the following command

```
string1, bool, string2 = struct.unpack('!8s?5s',data)
```

Note that in python3 struct.pack takes bytes objects instead of strings. You can turn your strings to utf-8 bytes with the command str.encode(string). Note that if you added parity to your string and encoded it the strings size increases. This is the reason why contents size is 128 instead of just 64.

UDP messaging

The program will initiate the UDP messaging by sending a message to the server. The message content should say "Hello from INSERT_CID_HERE", for example "Hello from ANjxnpwW".

The message content received from the server is a list of random words. The program must reply to the server with the words in reverse order. For example, if the received message is "correct horse battery staple", the program must reply with "staple battery horse correct".

Note that the lists of words the server sends are at maximum 64 bytes long, even though the content field is 128 bytes long. This is due to encoding of character with parity increasing the strings size. Rest of the content field is padded.

If the program worked correctly, the final message from the server will contain the number of replies and features used, e.g. "You replied to 6 messages with 3 features. Bye.".

Extra features

In addition to the baseline defined above, you can add some extra features for bonus points. Some extra features are mandatory for groups. The use of extra features must be negotiated during the TCP handshake.

Encryption

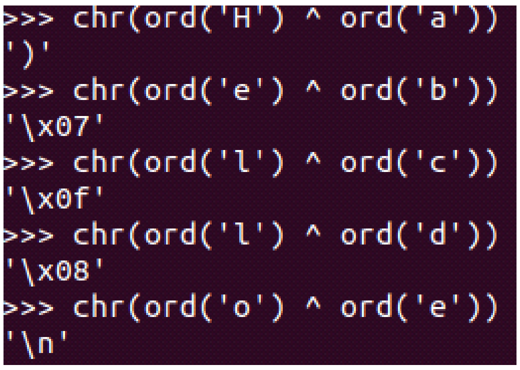
In this feature you will communicate using encryption. We will use so called one-time pad encryption, where client and server exchange list of keys and use one key for encryption only once. After a key has been used it must be discarded. You might question the exchange of keys over plain text TCP and you'd be right to do so. This is an oversimplified example for the sake of this exercise.

- 20 keys are exchanged in the TCP handshake
- The line after the last key must be ".\r\n"

- The keys are 64 byte strings of hexadecimal characters
- Encryption and decrytion is done by XORing the numeric value of each character in the message with the numeric value of the corresponding character in the key
- Sent messages are encrypted with the keys generated by the program
- Received messages are decrypted with the keys received from the server
- Each message is encrypted with a new key, keys are never reused
- The initial "Hello from" UDP message must be encrypted
- The last message from the server (when the EOM bit is set) is not encrypted
- If the keys run out during messaging program should switch to plain text messaging

Example

Given the message “Hello” and key “abcde”, the encrypted string is “)\x07\x0f\x08\n”



```
>>> chr(ord('H') ^ ord('a'))
')'
>>> chr(ord('e') ^ ord('b'))
'\x07'
>>> chr(ord('l') ^ ord('c'))
'\x0f'
>>> chr(ord('l') ^ ord('d'))
'\x08'
>>> chr(ord('o') ^ ord('e'))
'\n'
```

Multipart messages

In default functionality, the messages are at most **64 characters long**. Each message therefore results in a one packet. Multipart messages have a longer length and are divided into multiple packets.

- When reading messages, the program must be able to read messages in multiple parts
- When sending messages larger than 64 bytes, the message must be split into chunks of 64 bytes and each send separately
- The data remaining field in the header tells how much data is left unsent. For example, given a message content of 128 bytes, two messages must be sent. In the first message, the remaining field is 64, in the second it is 0.

```
def pieces(message, length=64):
    return [list of message pieces]

remain = len(message)
for piece in pieces(message):
    remain -= len(piece)
    data = pack(piece)
    send(piece)
```

Parity

Parity checking adds an even parity bit to each character in the message content. The even parity bit is 1 when the count of 1-bits in the characters value is odd, making the total count of 1-bits in the whole value even.

To add the parity, for each character in the content:

- Calculate the parity bit of the numeric value of the character
- Left shift the character value by one bit
- Add the parity bit to the character value

When using encryption, the parity must be added after encryption.

To check the parity, for each character in the content:

- Read the parity bit from the numeric value of the character
- Right shift the character value by one bit
- Calculate the parity of the character value and compare to received parity

When using encryption, the parity must be checked before decryption. The last message from the server (when the EOM bit is set) does not have the parity.

The server will send some messages with incorrect parity. If the parity check fails:

- The program must send a message to the server with the ACK bit set to false and with the message content "Send again"
- The server will then re-send the previous message

Note: The server will send all parts of a multipart message before reading the reply from the client, therefore if parity check fails, the program should read all the message parts before sending the "Send again" message.

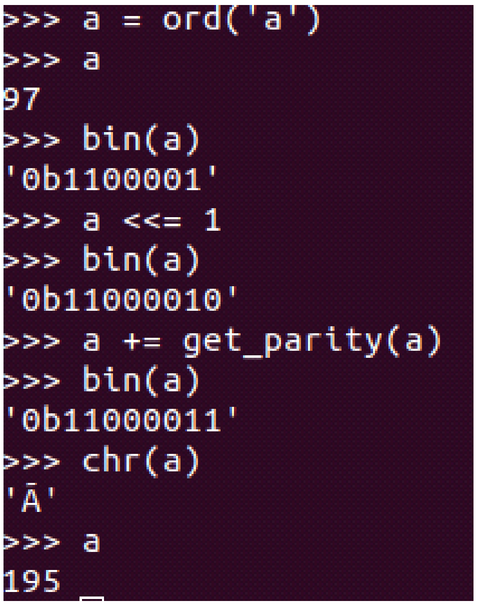
Note 2: When you encode your message with parity its size increases. Content length in the UDP packet should match the length before this size increase. Basically, take the length of your message before you encode it.

Example

The following python code returns the even parity bit of the given number:

```
def get_parity(n):
    while n > 1:
        n = (n >> 1) ^ (n & 1)
    return n
```

The following example adds the parity bit to the character "a":



```
>>> a = ord('a')
>>> a
97
>>> bin(a)
'0b1100001'
>>> a <<= 1
>>> bin(a)
'0b11000010'
>>> a += get_parity(a)
>>> bin(a)
'0b11000011'
>>> chr(a)
'Ã'
>>> a
195
```