

Faculty of Engineering

Electrical Engineering Department

**Project: Small Online Game**

COMP 454 / COMP 580

Spring 2020 – 2021

By:

Khaled Shazbek – 201903085

Khaled Damaj – 201800748

Mariam Manana – 201901873

Omar Kabbani – 201901288

Rawan Zahed – 201901907

Mohammad Itani – 201900527

# How to run the Project:

1. Install PyCharm or any other IDE of your choice.
2. First, open the project from the IDE and **run attacker.py**.
3. Second, open another terminal and **run defender.py after** starting the attacker script.

# Introduction

The main aim of this project is to develop a small multiplayer game. The project is done by implementing socket programming to connect two nodes on network so that they communicate with each other and are able to send and receive data.

# Description on the topic

The game has an attacker and a defender. In a match consisting of several rounds, a player is either an attacker or a defender, but not both at the same time. The game aims at sending an ATTACK message from the server to the defender which tries to scramble or ruin the message

# Implementation

For the attacker and defender to communicate and send messages, a socket API is used to provide an inter-process communication. The attacker initializes a TCP connection by creating a socket, binding to the port and then it sends a listen waiting for the defender to connect.

Thus, first we import the socket library which allows for socket programming:



A socket is created using the above command. The arguments passed to **socket ()** are the following:

**socket.AF\_INIT:** refers to the address family IPv4

**socket.SOCK\_STREAM:** refers to connection-oriented TCP protocol

This now creates the basis for the connection between the two hosts.

The game is built based on a server-client program, where the server is the attacker and the client is the defender.

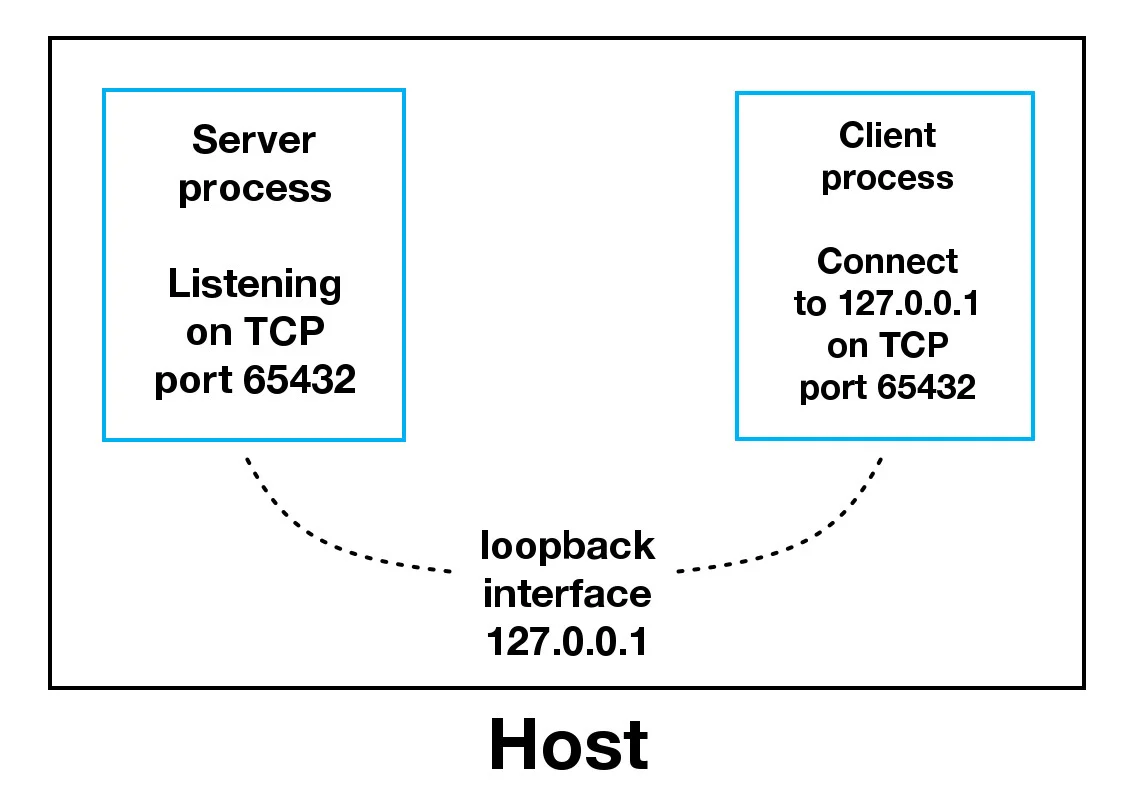


Figure 1: Client-Server Programming

## Attacker (Server-side):

A server has a **bind()** method to associate with a specific IP and port so it can listen to incoming request on that IP and port. It also has a **listen()** method which puts the server into listen mode while it waits for the client to connect. The **accept()** method accept TCP client connection.  
The number 5 in the **listen()** method refers to the queue size allowed before it starts blocking incoming requests.

Graphical user interface, text, application

Description automatically generated

## Defender (Client-Side):

The below scrip is self-explanatory. It shows how any basic client would work

Graphical user interface, text, application

Description automatically generated

This simple server-client program is the base on which the game is built.

## Hamming Code (Attack/Defense Algorithm):

The error correction algorithm is the hamming code. The hamming code will be implemented in the game to: first, encode each character in the ATTACK message to be sent by inserting redundant bits within the each character in the message. These bits are inserted at specific position to enable error correction and detection. At the receiver side, the receiver will perform a recalculation to detect error and find the bit position that has the error and correct it..

It is to note that the message that will be send will be converted into asci-binary before being encoded. This means that every character is represented by a bit sequence of 6-7 bits. For ease of decoding the sequences are stored in a list item, and the list containing all bit sequences is sent in an object using python pickles to the client where it will be handled.

Hamming code is capable of **detecting up to 2 errors simultaneously and to correct a single-bit error**. Thus, on the receiver side, after the message is received, there is also an error checking function which checks and tries to correct errors that it can find in the encoded message. Lastly the hamming code also is used to decode the already generated hamming code to be able to convert it back to characters (This method will convert a bit sequence with redundant bits back to its original bit sequence.)

So, how does the used hamming code exactly work?

**The encoding algorithm at the sender side:**

The sender encodes the message by following the steps below:

1. **Calculation of the number of redundant bits:** by using the **2p ≥ m + p +1** where m is the number of data bits of the message and r is the number of redundant bits should be added. For example:

The number of data bits are 4, by using the equation we will find the p should be equal to 4 (8 ≥ 4 +3 +1), so the total number of bits will be 7

1. **Position the redundant bits:** by placing these bits at the position of powers of 2 (1, 2, 4, 8, 16…). They are referred in the reset of this report as p1 (at position 1), p2 (at position2), p4(at position 4), and so on.
2. **Calculating the values of each redundant bits:** each redundant bit is calculated as the parity generally even parity. The following positions are defined in the algorithm itself:

* P1 is the parity bit for every data bit in positions whose binary representation includes a 1 in the less important position not including 1 like (3, 5, 7, 9 …).
* P2 is the parity bit for every data bits in positions whose binary representation include 1 in the position 2 from right, not including 2 like (3, 6, 7, 10, 11…)
* P3 is the parity bit for every bit in positions whose binary representation includes a 1 in the position 3 from right not include 4 like (5-7, 12-15…)

**Example**

Suppose the total bits of the message is 4: (1011). So, m =4.

This means we need 3 redundant bits (8 ≥ 4 +3 +1), so p=3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 |  | 1 |  |  |
| D7 | D6 | D5 | P4 | D3 | P2 | P1 |

D will represent the bit of the message and P the redundant bit

P1 will depends upon D3, D5 and D7:

* D3 = 1
* D5 = 1
* D7 = 1

The total number of 1 is odd, but p1 is calculated as even parity so P1 = 1

P2 will depends upon D3, D6 and D7:

* D3 = 1
* D6 = 0
* D7 = 1

The total number of 1 is even, which is correct so **P2 = 0** P3 will depends upon D3, D5 and D7:

* D5 = 1
* D6 = 0
* D7 = 1

The total number of 1 is even, which is correct so **P4= 0**

So, the encoded message will be **1010101**

**The decoding algorithm at the receiver side:**

Once the receiver gets the message, it performs recalculation to detect errors and correct them by using the following steps:

1. **Calculation of the number of redundant bits:** by using the same formula above.
2. **Positioning the redundant bits.**
3. **Error detection by using parity check**
4. **Error correction**

**Continuing the example above:**

Suppose noise has been added in the channel, an error bit will occur. Thus assume the transmitted bits are 1**1**10101(without noise 1010101)

**First,** the receiver will get 1110101. **Next**, it will calculate the number of redundant bits, by using the above formula, it finds that the total number of redundant bits are 3. Then it will position the redundant bits and it will get the following:

* **P1 = 1**
* **P2 = 0**
* **P4 = 0**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| D7 | D6 | D5 | P4 | D3 | P2 | P1 |

**Error detection and Correction**

Next, it will check the even parity of each redundant bit based on the data bits using the same rule for the generation of p1, p2, and p3 and then it will correct the error.Thus,

P1 depends on D3, D5 and D7:

P1 = 1

* D3 = 1
* D5 = 1
* D7 = 1

The total number of 1 is even which is correct. So, it set **P1 to 0** because it is correct.

P2 depends on D3, D6 and D7:

P2 = 0

* D3 = 1
* D6 = 1
* D7 = 1

The total number of 1 is odd, so there is an error. So, it set **P2 to 1.**

P4 depends on D3, D6 and D7:

P4 = 0

* D5 = 1
* D6 = 1
* D7 = 1

The total number of 1 is odd, so there is an error. So, it set **P4 to 1**.

Thus P4P2P1 = (110)2 = (6)10. This means that the bit of position six has the error so it should be 0 not 1.

**Before detecting and correcting the error:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| D7 | D6 | D5 | P4 | D3 | P2 | P1 |

**After detecting the error in position 6, we correct it by replacing 1 to 0:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| D7 | D6 | D5 | P4 | D3 | P2 | P1 |

**Hamming code reference:** [*https://github.com/danielmuthama/Hamming-Code/blob/master/hammingcode.py*](https://github.com/danielmuthama/Hamming-Code/blob/master/hammingcode.py)

Now let’s go over what is really happening in the game:

## Game:

The attacker will open a connection and wait for the defender.

The attacker sends the ATTACK message through invoking the designated function “sendMessageEncoded (clientsocket, message, timestamp)”.

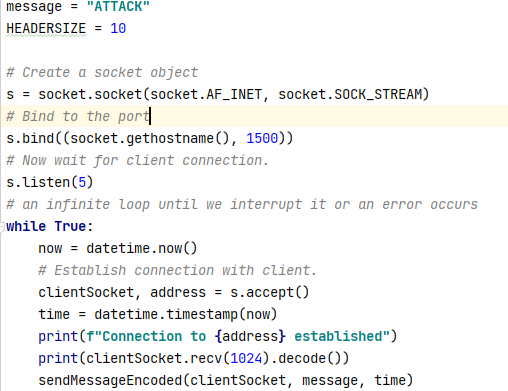
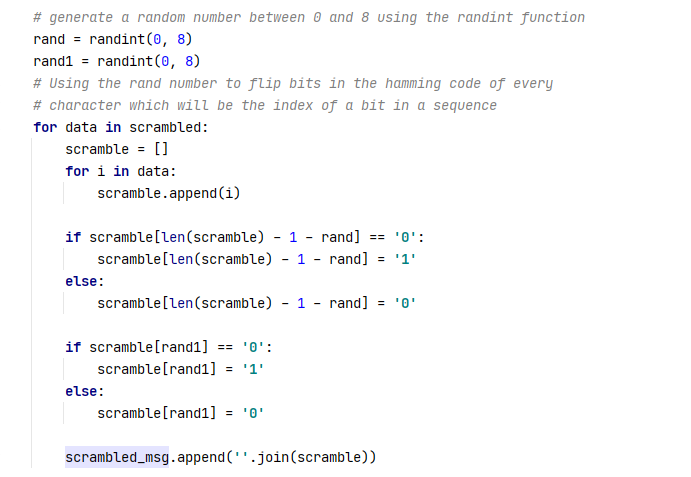


Figure 2: Attacker-side

In the function, the message first is encoded by generating a hamming code, then an error is introduced to the encoded message by flipping a random bit in each character of the message, twice.



The scrambled message and a timestamp that is used for scoring purposes which we will discuss later is then stored in a dictionary. The dictionary and an added header are then sent to the defender as a pickled message

Messages usually don’t arrive in one part, they may arrive in many stages. This is why the defender algorithm checks if the message is complete before handling it. The check happens through checking the length of the message. If the message has arrived completely, and before the timeout that’s decided based on average round trip time occurs, it is checked for errors then decoded. However, if the timeout occurred, the result of the round is immediately in favor of the defender, so the further steps are skipped and the attack has failed.

On a side note regarding the round trip time, it measures the time between the opening of the connection and the receipt of the message. However, since the first timestamp is on the attacker side and the other is on the defender side, this means that the first round is definitely going to be in timeout as the time actually extends while the user runs the defender code.

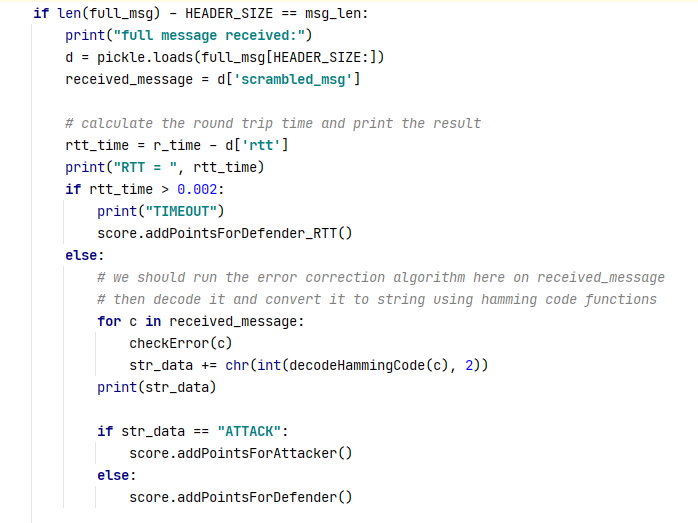
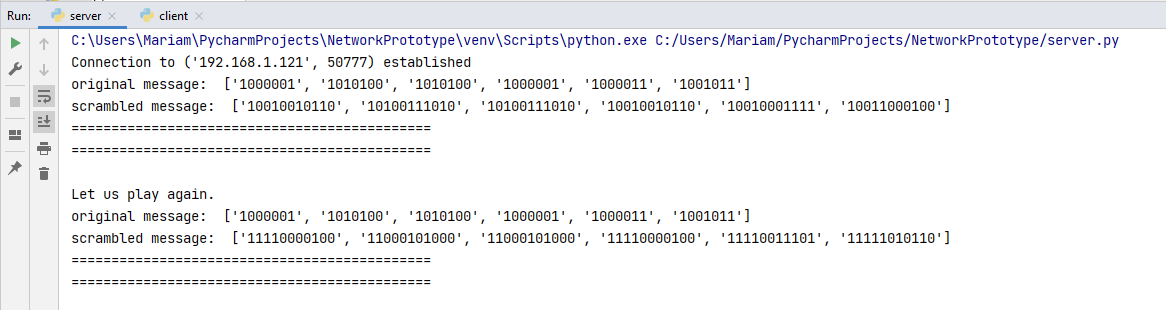


Figure 3: Defender-side

Checking the message for errors is an attempt to let the attack succeed, but the hamming code error-check algorithm is a single bit error check, so having an even number of bits flipped will not be detected, and having an odd number of bits flipped will only lead to one of them being corrected. This keeps the uncertainty of the attack which is needed for the game.

After decoding the message, we finally check if the attack succeeded and the score is modified based on the result of the round.

The attacker output will show a message telling that the connection to the defender has succeeded. It will also show the original message in binary and the version that has been encoded and scrambled and actually sent. For a match of two rounds:



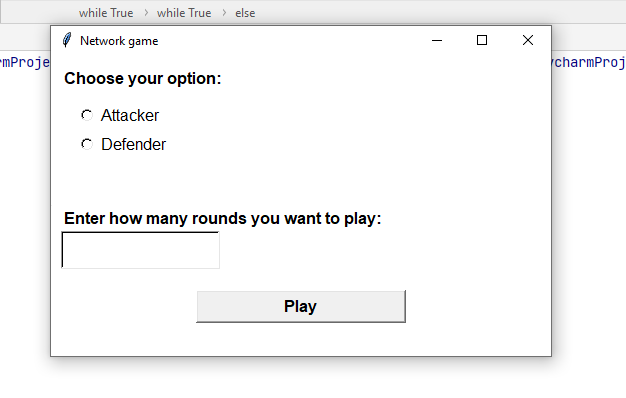
The defender side will show a message telling that the message has been fully received. It will also show the round trip time, the content of the received message or the word TIMEOUT when there’s a timeout, and the cumulative score of the defender and the attacker.



This is how the essence of the game is built. The remaining parts are all related to how the GUI functions, and how a user can play the game.

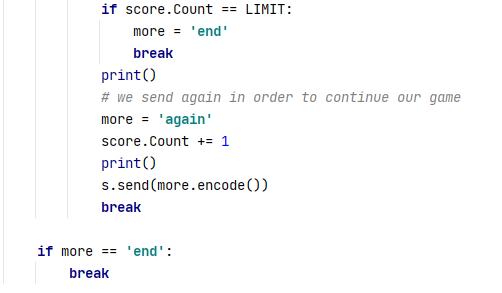
## Gamification:

Once the defender code runs, the user will be asked to choose which side he/she wants to play as, and how many rounds does he want the match to be.



This is implemented through using the functions of the tkinter library in python for opening and designing the window. The two options are kept as numbers 1 and 2 in list to help keep track of the choice and make sure that the user enters a choice before playing. The play button invokes a run() function that activates the defender code.

Since the user enter the number of rounds he/she wants to play, the connection between attacker and defender closes when the number of rounds is reached. As long as there are more rounds to play, the defender sends a message saying “again” to the attacker asking it for another round.



Once the connection closes, the user will get a window showing the scores of defender and attacks and telling him/her either “YOU WIN!”, “YOU LOSE!”, or “IT’S A TIE”. The window is also implemented using tkinter library functions. The details of the implementation are found as comments inside the .py file.

