Computer Science Programming Project 2020/21

Demonstrating Deep Packet Capturing and Analysis

Bailey Findlay

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Table of Contents

[**Table of Contents**](#_60slfzfwaok0) **2**

[**Analysis**](#_szx2k5qzhm5w) **3**

[Introduction and Problem Definition](#_flhulosuzxdn) 4

[Intended Users](#_14jn1t4jwlkv) 4

[**Research**](#_mys6gkzjnpj) **5**

[Language Suitability](#_coszk2jf40n2) 5

[Wireshark Analysis](#_bens07bzg6j8) 6

[Professional Feedback](#_hinrwrd27qd2) 8

[Networking Layer Models and Encapsulation](#_mtnrsr6quej2) 10

[**Data and Algorithms**](#_mvgq27fd3u7c) **14**

[Data](#_xoby5tvhgqrc) 14

[Algorithms](#_krr0db73sfm) 16

[Model of Proposed System](#_t3mmrx37fh9) 18

[**Objectives**](#_ohm4doponmv9) **22**

[**Design**](#_wzg533usaqva) **24**

[**System Overview**](#_70pnadlqm5p4) **25**

[**Capture**](#_vnbb7p8w2egk) **26**

[**Enhanced Packet Analyser**](#_sgpqkwh87h4p) **29**

[**GUI**](#_zb52tz9zts7v) **43**

## 

## 

## 

## 

## 

## 

## 

# Analysis

## 

## 

## 

## 

## Introduction and Problem Definition

My project is an educational program that visualises and demonstrates exactly what and how data travels across a network. The need for this project comes from a problem that there is no educational application that looks at this level of networking in an effective way. The project aims to use real-time data and examples to aid in learning. I’m aiming to visualise the concepts surrounding networking relating to the sending and receiving of packets across one or more networks and I am aiming to allow a deep analysis of the captured network data.

## Intended Users

The project is fundamentally an educational one and therefore my aim was to create an application with which members across the educational community could use to support their studies. Therefore my project’s target audience are interested in network programming, cryptography and networking in general. Initially, before efforts were made to develop this application, I discussed how I should solve this problem with my Computer Science teacher who is a part of my target audience. I asked him questions, a few of which were: “*How should I visualise network packet sniffing?*”, “*What kind of technology would be the most effective to capture packets?*”, “*How can I allow my target audience to interact with my application?*”, “*What should the main functions of this application be?*” and “*How can I maximise the educational aspect of my application?*”.

After some discussion, a few key points were raised:

* I should use network sockets to capture data as they are the rawest form of network connection available to me in a high-level language.
* Python is the language I should use as it has a flexible implementation of network sockets, allowing extended capabilities when it comes to capturing network data.
* The main focus of my application was to aid in education. My user interface had to be appealing but functional and interactable. My application should follow a step-by-step process logical to the intended user. I should minimize the amount of data being “thrown” at the user at one time.
* The majority of networking concepts that I introduce to my application should be simplified and shown to the user to maximise transparency and minimise confusion. This includes concepts like the TCP/IP stack and IP headers.

## 

## 

## Research

My research started by analysing the industry standard packet sniffer ‘WireShark’. I downloaded the application and started reading through the manual and the documentation. After watching and reading numerous videos and papers and websites explaining the technology of packet sniffing, I wanted to first give myself the opportunity to program a socket connection in Java as that would form the base for my application.

## Language Suitability

Below is a simplified diagram of how my initial program worked:



A massive issue I encountered whilst socket programming in Java, is that the sockets were not picking up *all* of the network data passing through a port. After doing some research into socket implementation in Java, I concluded that it is impossible to open what is known as *promiscuous* sockets in Java. Java sockets are end-to-end implementations, they cannot listen on the network port for *all* traffic. However, Python allows entirely promiscuous sockets to be opened by use of the Windows winsock2.h WSAIoctl function (written in C).

Here is where I made the decision to change my language to Python, as it best suited this project.

## 

## Wireshark Analysis



Wireshark Program Diagram

Wireshark is the industry standard network packet sniffer and so researching how it works is very beneficial to the development of my application.

**GUI**

Handles all user input and output.

**Core**

Main code that holds and connects the other blocks.

**Epan**

Enhanced packet Analyser - the engine that analyses each packet, it also provides the

following APIs:

Protocol Tree (dissection information for an individual packet)

Dissectors (Various protocol dissectors)

Dissector Plugins (Support for implementation of dissectors individually)

Display Filters (Display filter engine)

**Wiretap**

Wiretap library is used to read and write capture files in libpcap.

**Capture**

Interface to the capture engine.

**Dumpcap**

Capture engine - this is the only function block that executes with elevated privileges.

**Libpcap**

External libraries that provide packet capture and filtering support across many

Platforms.

The main components that will make up my application are a custom programming of the Core, GUI, Capture and parts of the Epan.

A key part of the wireshark components is the display filters, to enhance the customisability and therefore the educational aspect of my application I will be implementing options for the user to filter the data they can see dependent on the different ports and protocols that are commonly used across networks.

## Professional Feedback

I decided to seek professional help from both a Digital Forensics law enforcement officer, and a private professional Digital Forensics investigator. The following is a transcript of the conversation I had with both, under aliases.

**Smorgan 22/08/2020**

*In general I would study the osi model and look at how wireshark identifies a given packet as an IP, HTTP, etc.*

*I would study encapsulation of packets and so forth*

**Kr Today at 11:15**

***@bailey*** *You'll most likely want to reconstruct TCP to be able to follow the many protocols built on top of it, otherwise analyzing each packet independently is very limited.*

*There are some quick & dirty workarounds of course, like most HTTP requests fit within one TCP packet but technically one could (and malware would) send "GE", "T / H" and "TTP/1.1" as separate packets for example instead of the usual "GET / HTTP/1.1", which would work perfectly fine on the receiving end but a firewall/IDS can be fooled by this.*

*You can see what I mean in Wireshark with the right click > Follow TCP Stream feature, or with tcpflow some.pcap on the CLI, and with LibNIDS as a library for C; I can't seem to find Java bindings for it but hopefully there is an equivalent.*

**bailey Today at 17:10**

*I've looked at the OSI model and done some prerequisite research into how wireshark works, the ethernet frame, how to identify packet structures and stuff, I'm trying to program this using vanilla libs where possible*

***@Kr*** *I really appreciate the feedback, though, what do you mean by "reconstruct TCP"?*

**Kr Today at 17:35**

***@bailey*** *Basically all protocols fall into one of two categories: datagram-based or stream-based; the former means that a single packet contains everything it needs (since the MTU is usually 1500 bytes there is plenty of room to fit things like a DHCP discovery, a DNS query, an ICMP ping, etc) while the latter allows the payload to stretch across packets (e.g. most images nowadays are more than a kilobyte, if not megabytes, so they will have to be transported piece by piece).*

**Kr Today at 17:45**

***@bailey*** *This introduces quite a few additional things to be considered: what if the pieces arrive in the wrong order, what if one is lost, what if they are being sent more quickly that the network or receiver can handle, etc. TCP addresses all of that.*

*A good approach to understanding this is to study the differences between UDP, DCCP, TCP and SCTP.*

*Basically:*

*1. All UDP does is put you in touch with a particular port, it doesn't handle any of the considerations above and its checksumming feature is pretty much useless. But this is great when you do not want flow control for example, which is the case of VPNs, hence the saying that "TCP over TCP is a bad idea".*

*2. DCCP is UDP plus flow control: it won't retransmit anything lost but at least it will try to regulate the speed, which is great for video streams for example: doesn't matter if you missed part of an image, you mostly care that it keeps going on instead of getting stuck (which is what TCP would do until it managed to retransmit that piece of image you don't care about anymore) and that the quality adjusts to make the most of your connection speed.*

*3. TCP as explained above.*

*4. SCTP is TCP plus the capability to handle multiple streams within one connection, which is especially interesting when you want to do multiple (see MPTCP) but that gets complicated quickly and is not very common.*

**bailey Today at 17:50**

*I see, so it would be a good idea for me to try to handle both datagram and stream based protocols within my application*

*in order to make my application efficient and actually functioning*

*So, a good idea would be for me to unpack data according to the protocol and category it belongs to, and to process data through my code that way*

*as to make it most sensical to the user*

*I think personally I want to start just by capturing and dissecting ICMP, TCP and UDP data*

A number of key points were discussed here:

* I should study packet encapsulation in order to properly implement algorithms to deal with the concept.
* It could be a good idea to use an object orientated approach as each packet should be self-encapsulated and have its own attributes.
* I should study the differences between different transport layer protocols and how their data varies both in appearance and transportation.

## 

## 

## 

## 

## Networking Layer Models and Encapsulation

All networks reflect a layered, logical progression that is based on the 7-layer OSI model. I researched heavily into this layered architecture as it helps identify the specific area in which I will be capturing my packets from. I evidently won’t be capturing packets at the Physical layer as it’s entirely abstracted. The most important layer for me in this project is the Data Link layer. It contains a reliable transmission of data frames between two nodes.



The OSI Model

From the Wireshark documentation, I learned that network packets exist in the Ethernet Frame which is a data link layer protocol data unit, in other words, a data unit on an Ethernet link transports Ethernet frames as it’s payload data. I thought the use of a diagram really helped me understand it.



The ethernet frame looks like this.

* The preamble is a 56 bit pattern of alternating bits, ending in 11, allowing devices on a network to understand that an ethernet is about to be received.
* The recipient MAC & sender MAC are addresses for NICs to identify devices.
* The Type is referring to which Ethernet frame which is being used, Ethernet II is the most common in use today as it's often used directly as the Internet Protocol. (Ipv4 or Ipv6).
* **The data section is the most important to us, it houses IP, ARP etc data as payload data. It’s an IP packet, it contains the data we want to capture and manipulate. It must contain between 46 and 1500 bytes of information.**
* The Frame Check Sequence is a cyclic redundancy check (CRC) that allows for the detection of corrupted data in the entirety of the frame (bar the preamble).

This level of information is really important with the designing of my algorithms as they must be able to manipulate the byte stream in correspondence with how an Ethernet frame is constructed in order to properly intercept and read the data. The OSI is a detailed, more common use implementation of our understanding of network structures, but the TCP/IP stack is what I’ll be implementing as it maintains all the aspects I’ll be using whilst simplifying the other components, which I believe will make it easier for my intended user.



Encapsulation

The most important level of networking that I will be directly implementing within the algorithms of my program is the abstract headers of IP, TCP and UDP. As seen in the diagram above. TCP and UDP data is part of the Transport layer. I will be unpacking packets from the reverse order of the TCP/IP stack as I’m deconstructing the packet, not constructing it.



TCP/IP Stack

Highlighted are the Network & Transport layer, these house the internetworking protocols (IP, ICMP and ECN) and the layered architecture protocols (TCP, UDP, SCTP). In order to understand the payload data, we need to understand which of these protocols each packet is using. I will be collecting only IP packets as they contribute 99% of the network traffic and from the IP header I can determine which Transport layer protocol each packet is using in the “Protocol” section.



IP Header

From this header I can also determine a lot of other useful information, I can understand whether or not the packet is IPv4 or IPv6, the headers length, it’s flags, its offset, the time to live, the source and destination IP addresses and its extra options.



TCP

(X is reserved and redundant in the context of my application)



UDP

All internet packets arrive within this format, first the IP header, then the UDP/TCP header, then the payload. Using this I can properly deconstruct the structure of a packet and extract its data.

It’s important to note that only TCP packets have Flags. This distinction will be made clear within my application as it’s significant to notice the difference between UDP and TCP and why these two protocols exist.

## 

## Data and Algorithms

The data and algorithms for my application can be expressed under the following main areas.

1. Data
   1. Packet classes including
      1. IP
      2. TCP
      3. UDP
   2. Payload data expressed as
      1. Raw expression data, hexadecimal
      2. Human readable data, unicode
2. Algorithms
   1. Capture raw network data
   2. Interpet raw network data
   3. Display interpreted network data
   4. Manipulate raw and interpreted data in interface
   5. Manipulate raw and interpreted data in storage
3. Multi-threaded

## Data

The majority of the data and algorithms will be expressed in a complex object orientated method of programming, and therefore it is easiest to represent these key data structures in object class diagrams and inheritance diagrams.



1.a.i. and 1.a.ii.

## 



1.a.iii., 1.b.i. and 1.b.ii.

Each of these classes inherits from a different class, as demonstrated in the inheritance diagram below.



The distinction between UDP and TCP packets is clearly shown by a different class and different inheritance path whilst maintaining its similarities in attributes and methods.

## Algorithms

2.a.

In order to capture raw network data, I’ll need to implement an algorithm that repeatedly receives data across a network adapter starting at the IP packet layer. My algorithm should be able to be toggled on or off, to capture or stop capturing. My algorithm will be able to properly manage each “chunk” of data (which are realistically bytestreams).

I have already discussed which type of technology I'll be using for this algorithm, a network socket connection. The algorithm will be solely responsible for the capturing of network data, and will pass on that data to our Packet classes.

The main data types that are dealt with in this algorithm are bytestreams in groups of 65,535 bytes. This data is then properly processed in 2.b.

2.b.

The main way I will be algorithmically interpreting the raw network data is by structurally unpacking the data by its bitcount. Using the network headers appended to the payload at each layer, I can properly and accurately dissect the raw network data into specific numbers that correspond to values of the header. For example the first four bits of the IP header corresponds to the version it is (IPv4 or IPv6).

My algorithms need to contain the idea of bit manipulation, I’ll be interpreting each byte as its own, but if a field takes up more than one byte it requires binary shifting. For example the Source IP address in the IP header takes up four bytes, this requires byte manipulation like this.



2.c.

My algorithm to display interpreted network data heavily depends on both the organisation of 2.b. (the classification of my data). Each packet will be its own object instance of the classes described in 1.a. and 1.b. this means my algorithm will display the specific object as clickable to the user, it should encapsulate the packet, the whole packet and nothing but the packet.

The algorithms to display specific parts of data from the packet will work exclusively with the objects presented to the user. Essentially the algorithm will simplify the massive amounts of data by dealing with one packet at a time when displaying specific data. This heavily aids my aim to maximise the understanding of my application and minimises any sort of confusion.

2.d.

A key part of my application is the ability to filter network data, to change and manipulate the data being seen by the user in accordance with what they wish to see.

My algorithms must be able to dynamically filter what the user sees depending on which filters they choose to enable. This includes being able to filter for multiple specific ports at a time and mixing filters.

2.e.

There will be algorithms to deal with the ability to both save and load captured packets from a selected file. Algorithms must be in place to make sure everything is smooth when saving and importing packets.

3.

My algorithms will, to an extent, be multi-threaded, a user-interface simply cannot update whilst I am looping and capturing data. I intend to make a main file where I handle all of my threads.

## 

## Model of Proposed System

My proposed system takes influence from Wireshark which I researched earlier. The main influence I took from Wireshark was the listing of packet objects. It looks like this:



The idea is that packets, when captured and have been through the process of interpretation, will populate a scrollable grid in which one row corresponds to one entire packet of information. I want each packet to be a clickable object, it should be encapsulated and independent of each other packet.

When a packet is clicked, all the information about this packet should populate three, appropriately titled and lower placed text boxes, “Hex”, “Unicode” and “All Information”.



Each textbox should be scrollable either through a scrollbar or a scrollable text window.

The idea is that a user can click one specific packet to inspect and analyse, and only one packet at a time. This should help minimise confusion for the user.

A major feature I wanted to include was a design graphic of the payload and header information. I want it to be populated with packet specific information when a packet object is clicked and it should look like this.



The aim is to help the user understand how a packet is structured using real-time examples.

I have left a distinct gap where I want my control panel to be. The control panel will be the area in which the user will be able to start and stop capturing, control all filters and launch the demonstrational server.

This is what I want my control panel to look like.



My server demonstration should either start a new page in the window, or a new window entirely, but it should interact with the original page as normal. The packet sniffer should exclusively pick up on this server demonstration when it is open.



In a basic setting the window/page should look like this. I think it should also clean the packet window so that all packet objects shown are captured from the demonstration. This way we don’t accidentally have both packets captured normally across the users network mixed with the demonstration window packets.

## 

## 

## 

## 

## 

## 

## Objectives

My project aims to allow my target audience to capture network packets, analyse them and manipulate them in order to learn more about the technology and theory behind networking.

I have made a comprehensive single sentence list of objectives I wish my project to achieve.

1. The user should be presented with an intuitive interface with clickable buttons and changeable screens.
   1. The screens should clearly state at which stage the user is capturing packets. (e.g home screen, select adapter, capturing).
2. The application should be customisable and entirely interactive.
   1. Each screen should display clickable buttons as well as changeable variables to enhance the way the user can control the application. (e.g changing protocol being scanned & changing filter options).
3. The user should be able to capture network traffic both intended and not intended for their current device.
   1. The application should, at base level, be able to capture network traffic in packet form.
   2. The application should be able to capture packets that are intended for the user’s device, as well as capture packets that are not intended for the user’s device.
   3. The application should be able to capture packets and save them in local storage.
   4. The user should be able to choose one or many protocols to capture from at any given time.
   5. The user should be able to select specific packets they have captured and be able to interpret the data they hold.
4. The application should be able to interpret network traffic.
   1. The user should be able to select which network adapter to capture traffic from.
   2. Network traffic should be, when encapsulated in packets, be interpreted as human-readable data, in this context, unicode.
   3. If the adapter 4.1 cannot perform 3.3 and by extension 3.4, an error message must prompt the user with further information.
5. The user should be able to save collected network data in packet form in a file on local storage.
   1. The user should therefore be able to load saved network data from a file on local storage into the program as if it were just captured.
6. The application should be able to connect to another application or itself via a client-server architecture.
   1. The user should be able to type messages to an automated self-hosted server.
   2. The user should have the option to use a custom port server.
      1. The user should be able to clearly see their own messages being sent and received by the server.
   3. The user should have the opportunity to capture traffic on the same port as the chat, and therefore demonstrate the capability of packet sniffing in that context.
      1. The user should have the opportunity to capture this traffic whilst it is using the custom protocol / port as well as any available port.

# Design

## 

## 

## System Overview

This application is complicated as it compiles different concepts and functions into one project so the actual structure of my program is very specific and well thought out. Both a diagram and an explanation of the structure will best layout my application.



**Main / Core**

Where the entire program is handled. It manages the threading of other modules like capture and the UI.

It houses the files/submodules:

“main.py”

**GUI**

Where all of the user interface is managed, it handles all the events and acts like a pipeline for the rest of the program to be presented to the user.

It houses the files/submodules:

“userinterface.py”

“serverdemo.py”

and the image subdirectory “img” to hold my assets

**Capture**

Where all of the network capturing is managed and maintained, this is the main part of the program as it handles the actual raw network data.

It houses the files/submodules:

“capture.py”

**Enhanced Packet Analyser**

Where the captured network data gets utilised and turned into readable data, it is the most complex module and persists with most of the employed technical concepts.

It houses the files/submodules:

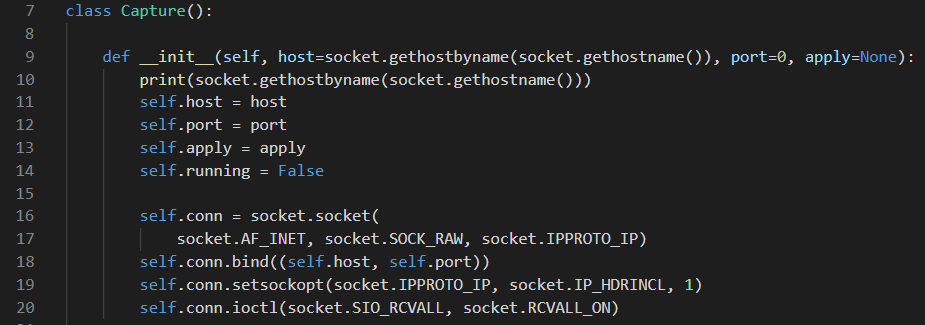
“packets.py”

“data.py”

## Capture

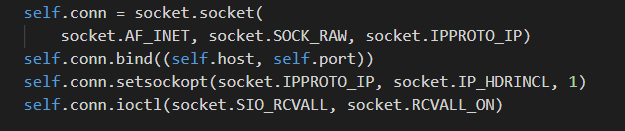
| **Capture.py members** | **Data Type** | **Purpose** |
| --- | --- | --- |
| Capture | Class | House the entire capturing operation so that I can objectify it. |
| host | Tuple | Defines the host address in an integer tuple, of IP format. |
| port | Integer | The port to capture network data on. Default set to 0 to capture all traffic. |
| apply | Function | Apply the result of the function to another function, in this class it does not take any shape. |
| running | Boolean | Whether or not the thread managing the capture object is running or not. |
| conn | Object | Socket connection with parameters. |
| start | Function | Creates a new thread in daemon mode and starts it, also sets the running variable to True. |
| stop | Function | Sets the running variable to false and therefore terminates the running thread. |
| \_loop | Function | Target function of the thread, it does the actual capturing and extraction. |
| raw\_data | Byte Array | The received raw data when capturing through a socket object. |
| packet | Object | Initialises the Packet class and creates a new packet for the data captured. |
| packet\_spec | Object | Initialises the specialised packet classes either TCPPacket or UDPPacket depending on its protocol. |

Logically the first place to start is where we get any of our network data from. The Capture module has a singular file “capture.py” and a singular class “Capture()”. It’s designed in a way for multithreading to efficiently manage the incoming data.



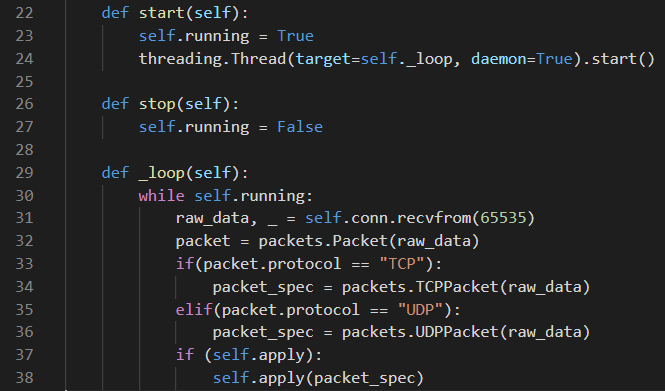
This part of my code initialises the class, it sets to receive information on a *promiscuous* socket. The host is set to the host’s local ip address and the port is set to 0, this means that data will be collected from *all* destination and source ports. This is important because the application is a sniffer, it should be collecting absolutely every bit of data it can.

The important part of this part of my code is where the socket is created and configured.



The first two lines create a socket connection using the IPv4 family in raw mode to collect IP packets. This means the program collects IP packets in version 4 which is important as it keeps our data uniform for the rest of the program.

I bind the socket to the predetermined host and port, then configure the socket to collect IP packets that explicitly already have an IP header, therefore we only get true packets, no passing data from the local user’s computer. Lastly, the socket is made *promiscuous*, it receives all packets with no exceptions.



This part of the Capture class is structured for threading, it has a flag to either start or stop the thread and a loop to collect data. It’s notable that I set the thread to *daemon* mode, meaning the thread will terminate when the master thread does (when the program closes).

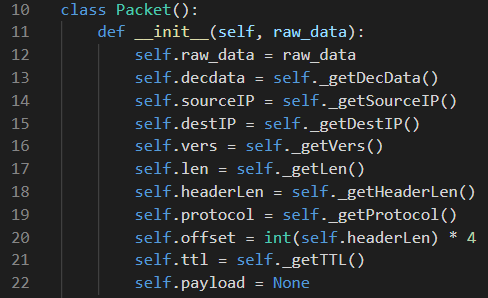
The loop collects data from the socket and passes it to the “packets.py” file in the Enhanced Packet Analyser module. A new object instance of the Packet class from “packets.py” is made and the protocol is checked against either “TCP” or “UDP”, this is because as I mentioned earlier in my research, they are separate protocols of the transport layer, so they have differently structured headers which results in a different programmatic approach to deconstructing them. It also means they require specialised objects which I also outlined earlier in my analysis.

## Enhanced Packet Analyser

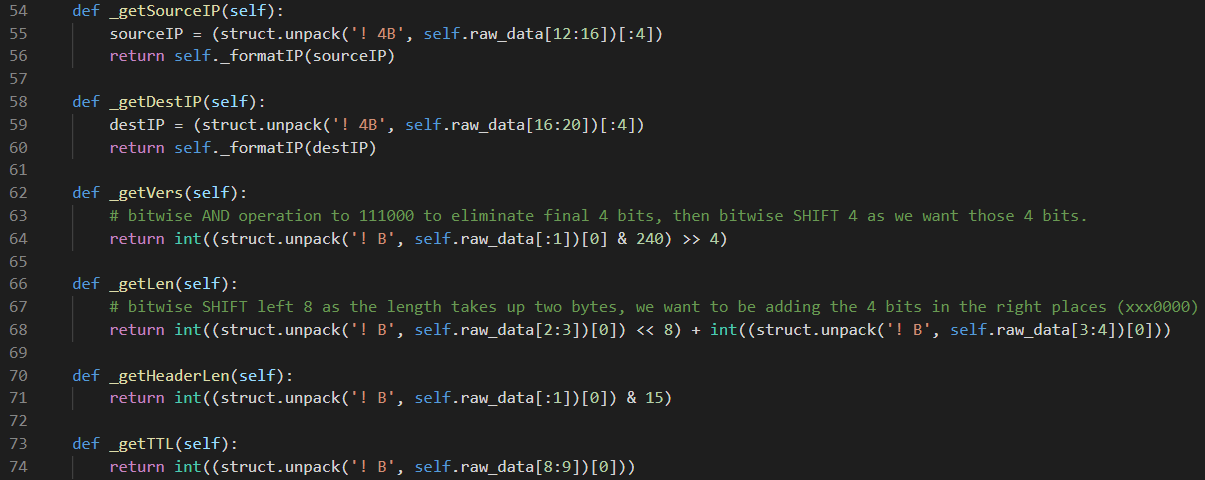
| **Packets.py members** | **Data Type** | **Purpose** |
| --- | --- | --- |
| Packet | Class | Objectify and encapsulate the concept of a packet for each entire unit of data acquired. |
| TCPPacket | Class | Differentiate and inherit from the typical Packet Class but with specialised attributes according to the TCP transport protocol. |
| UDPPacket | Class | Differentiate and inherit from the typical Packet Class but with specialised attributes according to the UDP transport protocol. |
| Payload | Class | Objectify and encapsulate the payload of a packet, inherits from both the Father Class (TCP or UDP Packet) and the Grandfather class (Packet). |
| raw\_data | Byte Array | The received raw data when capturing through a socket object. |
| dec\_data | Array | The decoded raw\_data in hex format. |
| sourceIP | String | The sender’s IP address of this particular packet. |
| destIP | String | The recipient’s IP address of this particular packet. |
| vers | Integer | The version of IP packet it is, usually 4 or 6. |
| len | Integer | Total length of the entire packet. |
| headerLen | Integer | The length of the IP header of the packet. |
| protocol | String | The specific transport protocol used by this packet. Usually TCP or UDP. |
| offset | Integer | The offset to start reading the transport layer header from. As to skip the already read IP header. |
| ttl | Integer | Time to Live, the amount of hops a packet has left. |
| payload | Object | Initialises the payload packet class to hold the actual payload data. |
| get\_json | Function | Returns the json dump of the decoded data. An optimised storage method. |
| \_formatIP | Function | Formats an integer tuple, exclusively an IP address, as a string. |
| \_getDecData | Function | Algorithmically decodes each byte of raw\_data into the dec\_data variable. |
| \_getSourceIP | Function | Calculates the source IP address as an integer tuple by C struct calculations and returns it. |
| \_getDestIP | Function | Calculates the destination IP address as an integer tuple by C struct calculations and returns it. |
| \_getVers | Function | Calculates the version of the IP packet as an integer by C struct calculations and returns it. |
| \_getLen | Function | Calculates the length of the entire IP packet as an integer by C struct calculations and returns it. |
| \_getHeaderLen | Function | Calculates the length of the IP header of the packet as an integer by C struct calculations and returns it. |
| \_getTTL | Function | Calculating the time to live, aka hops, as an integer by C struct calculations and returns it. |
| \_getProtocol | Function | Calculates the transport protocol used by the packet, and formats it as a string acronym, e.g TCP, and returns it. |
| srcPort | Integer | The port in which the packet was sent from in the sender’s network. |
| destPort | Integer | The port in which the packet was received from/sent to in the recipient’s network. |
| seqNum | Integer | The sequence number of the packet, this signifies a few different things based on what the number actually is. It’s the packet number in a sequence of multiple packets. |
| dOffset | Integer | The offset of the IP header and the TCP/UDP header. |
| flags | String/Tuple | Tuple (converted to string) that gives the values of all of the flags attached to a TCP header, note this doesn’t exist in the UDP class. |
| payload | Byte Array | The actual data holding the payload of the packet, aka the packet stripped of all headers. |
| \_getSrcPort | Function | Calculates the srcPort variable by C struct calculations and returns it. |
| \_getDestPort | Function | Calculates the destPort variable by C struct calculations and returns it. |
| \_getSequenceNum | Function | Calculates the the seqNum variable by C struct calculations and returns it. |
| \_getTCPHLen | Function | Calculates the the header length of the TCP header variable by C struct calculations and returns it. |
| \_getFlags | Function | Calculates the the flags variable by C struct calculations and returns it. Importantly, these are separate calculations for each flag, but combined at the end for ease of use. |
| \_getUDPHLen | Function | Calculates the header length of the UDP header variable by C struct calculations and returns it. |
| payloaddata | Byte Array | Calculates the offset raw\_data variable so that the headers are stripped. |
| data | Array | Holds the raw\_data variable in hex format via a different calculation than the decdata variable. |
| Data | Class | Objectify and encapsulate the whole operation of saving, managing and loading files of packets. |
| date | String | Current date, in the format Day/Month |
| createFile | Function | Creates the new file whenever packets are saved. |
| directory | String | The date as a string. |
| parent\_dir | String | Holds a default path, C:/Packet Captures/ as a parent directory. |
| path | Object | OS path object consisting of the parent\_dir and directory variables joined together |
| file\_path | String | The file’s full path with extension, “.json”. |
| n | Integer | The number of files already in the current date’s directory. |
| loadFile | Function | Handles the loading of a file whenever packets are to be loaded. |
| data\_list | Array | Holds all of the individual packets that have been saved in the file. |
| normalline | Bytes Array | Holds the bytes array of the packets, sequentially. |
| newPacket | Object | Creates a new object of the Packet class for each previously saved packet. |
| get\_normal | Function | Strips a line read from a file so that it contains only the necessary data. |

The Epan module contains two files, “packets.py” and “data.py”. “packets.py” is where the raw network data is actually deconstructed into our abstract formats like the TCP/IP stack.

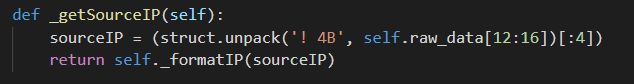
There are four major classes in this file called Packet(), TCPPacket(), UDPPacket() and Payload().



The initialisation of the Packet() class is called whenever we get a new chunk of raw data, which are obviously already sent in packets as they are internet packets of data. The purpose of this initialisation is to simply call its private methods and calculate the relevant information. There is one method per bit of information we need in relation to the IPv4 header of the packet, which I explained in my research.



The majority of these classes boil down to bit manipulation of the raw data in order to obtain the correct values.

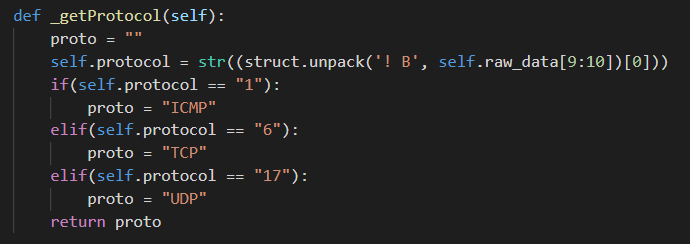


For example, to get the source IP, we need 32 bits starting from the 96th bit. This is where the source IP address is stored when the packet is constructed.

I use the python struct function which performs conversions between python values and C language structs. It’s how I can actually interpret the bytes and packed binary data.

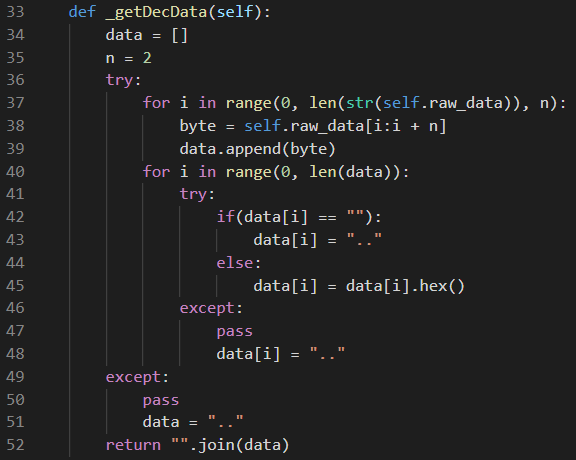
“struct.unpack(‘! 4B’ “ means I’m unpacking four bytes of data, “self.raw\_data[12:16])[:4])” means I’m taking exactly four bytes of the raw data between the 12th and 16th bytes, or 96th to 128th bit. This is how I exactly calculate the real number.

The methods that calculate the version and length use bit manipulation in order to obtain the correct value. For example the length spans two entire bytes, but because we interpret this total length as one integer the program needs to perform some arithmetic with two separate bytes, which also means shifting the bytes. I explained this problem earlier in my analysis. I use very specific bit manipulation like << (shift right) and & (bitwise AND operation) in order to obtain the correct value, the whole value and nothing but the value.



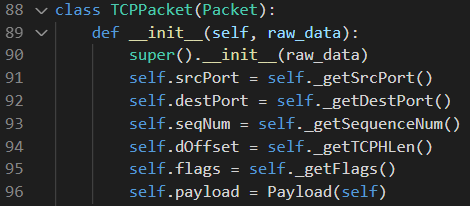
The Transport protocol of the Packet is also decided in this class, I make the distinction of assigning the protocol variable to the text name of the protocol rather than the number, it makes displaying and processing the Packet’s protocol more efficient down the line.

Importantly, I algorithmically decode the payload section of our data. This is also done in the Payload() class, however it’s important that the calculation is done when the first class is initialised as it helps with the setting up of the user interface.

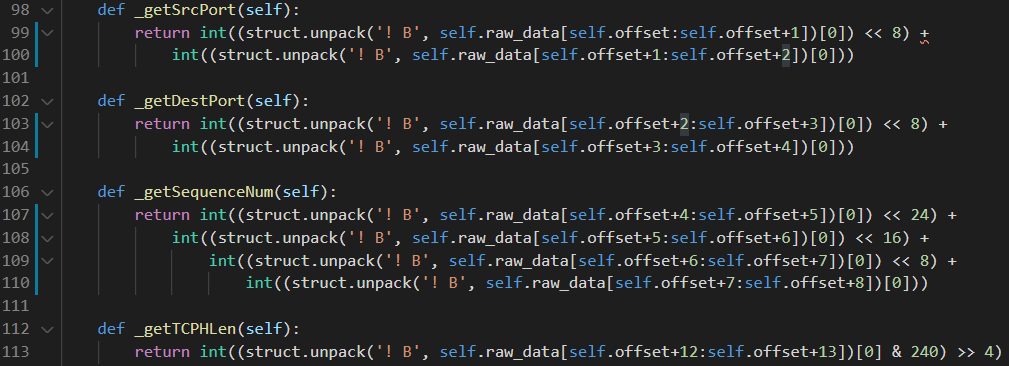


In this method, the program uses iteration twice, once to split the raw data into bytes, and once to actually decode the data. It’s important to do this because the data is constructed like this, if we tried to decode the entire string all at once the outcome would be incorrect. Like the entire program, it’s important to consider how the packets are constructed in order to deconstruct them correctly. I also have a try catch statement as not all of the constructed data can be decoded as it's often encrypted, meaning we get nonsense data as a result. Instead of displaying nonsense data I replace it with two dots representing two nonsense bytes. The reason this is done so early in the program is that it helps with storing any packets the user wishes to.

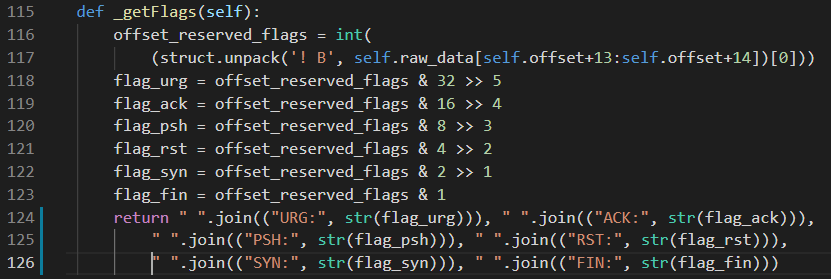
In the same file TCPPacket() exists, it has a very similar structured initialization file.



TCPPacket() takes the earlier Packet() class as a constructor, it inherits all of the structure and information from Packet() as it is treated as a “specialised” packet. It’s important to keep track of all of the information a singular packet has, so the specialised packets need to inherit from the non-specialised packet class. Notably, the initialization also creates a new payload class which I’ll get to later.



The methods follow a similar path to Packet() with the exception that, the options I calculate, like the sequence number, are often very large numbers and need more complex bit manipulation and binary byte splitting. The TCP sequence number spans four bytes, so it needs



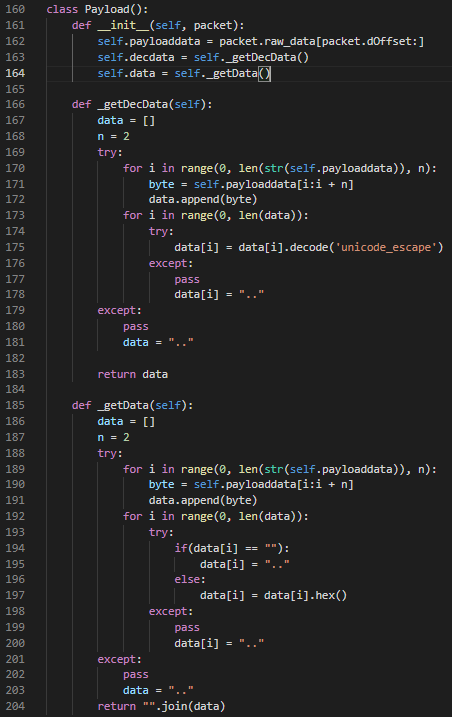
The flags of the packet are also calculated here as if the Packet is a TCP packet, it must have flags. These flags also determine other factors of the packet’s nature and size so the calculations are 100% accurate. For example, if the SYN flag = 1, then then it indicates that this is the first packet in a sequence. If the FIN flag = 1 it is the final packet in a sequence from a sender.



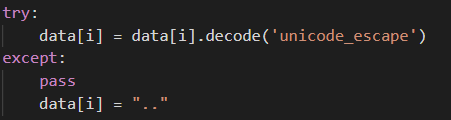
UDPPacket() is the other programmed class of specialised packet, it takes Packet() as a constructor and inherits all of the above properties much the same as TCPPacket(). It also creates a payload object.

The program is designed so that each Packet cascades it’s own specialised packet class and that packet class cascades it’s own payload class. This way I keep data to a complete nominal form where I can only access the data necessary rather than all of the data.

The algorithms here are bespoke for the UDP header. They specifically calculate the relevant attributes such as Destination Port and Source Port in relation to how UDP communication is designed.

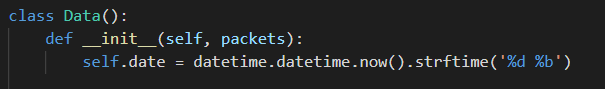


To conclude the “packets.py” file, the Payload() class handles the actual payload data being transmitted across the network. It strips down the headers in sequence of the TCP/IP stack and decodes the bytes sequentially. I do this twice to handle both the decoded data and the raw data, I need this later when I present both to the user.

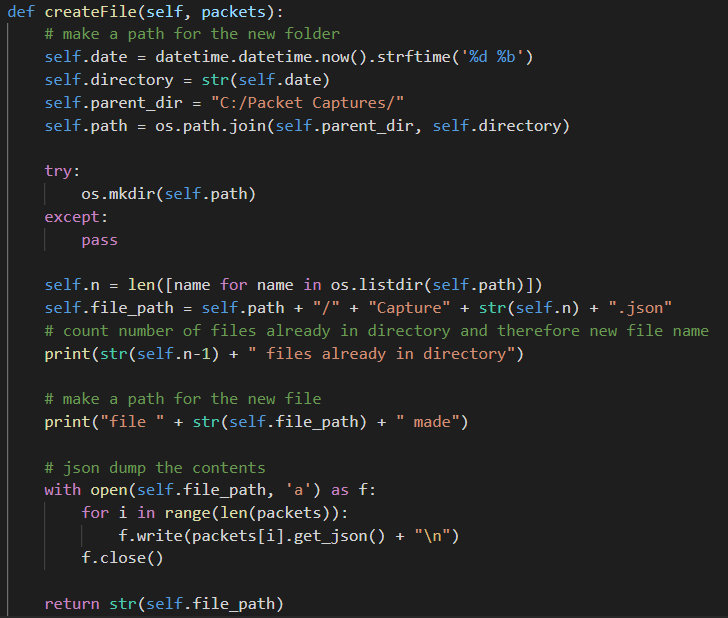


The first algorithm decodes using escaped unicode which is the only unicode encoding that properly interprets the network data being captured. It’s principally just unicode encoding with the important part being the escaping of characters, meaning some bits are treated and decoded differently depending on the surrounding characters.

“data.py” is the file that handles the entire operation of saving, writing, reading and loading packet data from and to files. There is one class in the file called Data().

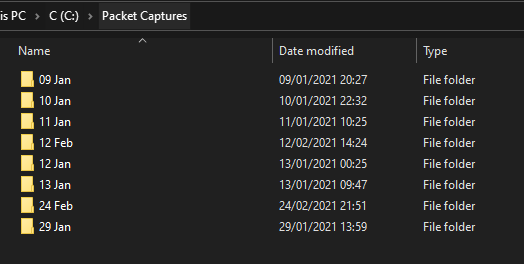


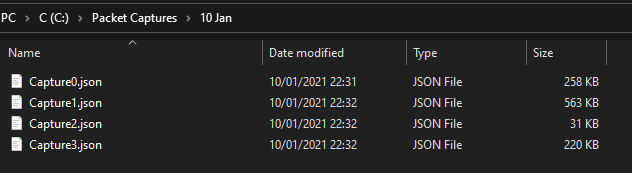
The date and time are initialised when the class is initialised and updated later as a check, in case the program is being run for an extended period of time.

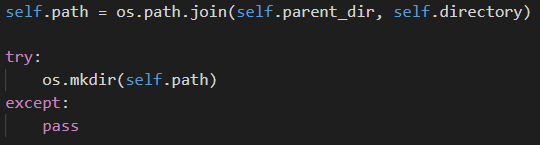


The function createFile() is called whenever the user decides to save the packets that are currently captured. I update the date to make sure it's accurate and stringify it as a new directory. The base parent directory is always set as C:/Packet Captures/, this will be the same among all users.

The result looks like this.

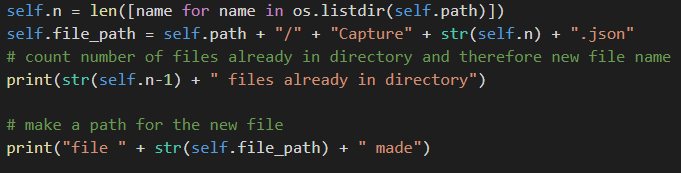




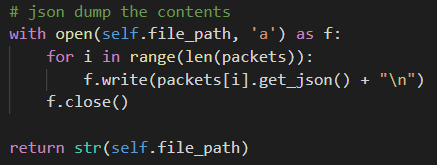


The program interacts with the operating system by creating a new OS path for the folder.

Then, the path is created for the exact file being made that houses the current captured packets.



The variable n counts the number of files already initialised that day, if it’s the first file then the name of the file will be “Capture0.json”.



The contents of each packet are then written to the file in sequential order via a get\_json() method from the Packet() class.

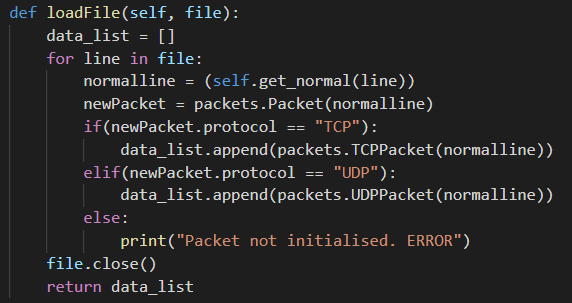


The function returns a json.dump of the decoded data. This is highly optimal as it’s a very low-cost function on the CPU of converting data to JSON format. It’s even more space efficient as we’re only storing unicode formatted data in which we can garner all of the needed information, instead of storing the raw data. A typical save file looks like this.

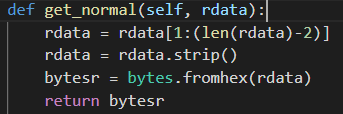


Note that the filename is returned so the program can inform the user of exactly where their saved packet capture file is kept.

The next function handles the loading of a saved file to the application.



The function reads the file in a typical way, it grabs the file as an object and iterates through each line. The iterative process uses another local function, get\_normal().



It removes null parts formed by the structure of multiple packets, it strips it of any white space and then converts the stored data from hex.

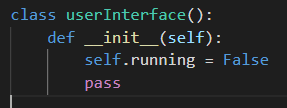
A designated line separator differentiates one packet from another, and a new packet object is created solely on the data in each line. Creating a new packet from the existing class template allows the program to utilise its algorithms already in place instead of having to make new ones. The function also cares to make a specialised packet and handle errors regarding the reading of the file.

## GUI

| **Serverdemo.py members** | **Data Type** | **Purpose** |
| --- | --- | --- |
| userInterface | Class | Objectify and encapsulate the entire interface regarding the server demonstration. |
| running | Boolean | Whether or not the thread managing the capture object is running or not. |
| usr\_message | String | The string message entered by the user to send to the server side. |
| server\_text | TkInter Object | TkInter text box to pack all of the relevant server information. |
| server | Object | Class object of the Server class. |
| client | Object | Class object of the Client class. |
| \_loop | Function | Target function of the running overall thread for the server demonstration. |
| stop | Function | Stops the server via another function, and set the userInterface thread to terminate. |
| onHover | Function | Button aesthetics. |
| onClick | Function | Calls the submit function when clicked. |
| Server | Class | Objectify and encapsulate the server side of the server-client demonstration. |
| ui | Object | An object reference to the userInterface class. |
| host | String | IP host of the server. |
| port | Integer | The port that the server hosts on. Default 7789, a very frequently unbound port. |
| server\_socket | Object | Socket object. |
| start | Function | Sets running to true, creates and starts a thread on daemon mode. |
| stop | Function | Terminates the server thread. |
| \_loop | Function | Target function of the Server thread, it handles the entire process of actually receiving data over the socket. It also handles packing it into the interface. |
| Client | Class | Objectify and encapsulate the client side of the server demonstration. |
| client\_socket | Object | Socket object on the client side. |

The GUI module contains two files, “userinterface.py” and “serverdemo.py”. “serverdemo.py” is the file that handles the entire operation of setting up, establishing connection, maintaining connection and closing connection of the client-server demonstration..

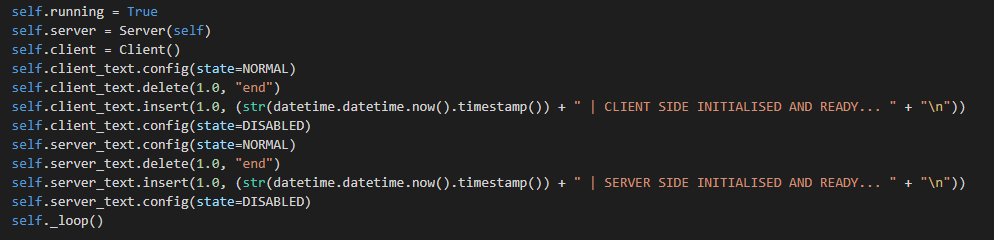
There are three major classes in this file called userInterface(), Server() and Client(). userInterface() creates the actual visual of the server demonstration for the user.



As it’s a functioning user interface, it’s threaded like the main user interface. I’ve programmed the threading specifically so that one sub-thread of the main thread can manage the server demonstration whilst another sub-thread can manage packet capturing. It’s a crucially significant part of the program that means it all runs together smoothly and functions as expected.

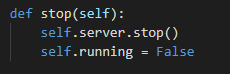


The userInterface() class has the significant function start() which is the target function of the user interface thread. I use tkinter to make the UI, start() packs frames to house data inside of each other which makes a modulated approach to user interface whilst making the housed data extremely maintainable via self-encapsulation.

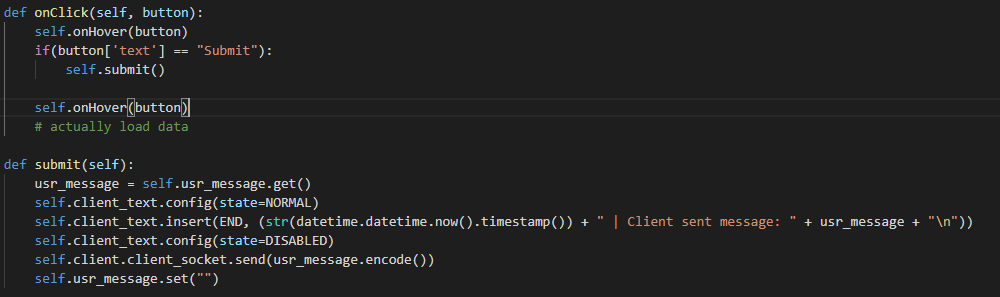


Most of this function is creating frames, labels, textboxes and such. The end of the function verifies the thread is running and starts a client and server object from the respective classes. It also initialises the textboxes with startup messages to let the user know it’s functional and ready to be used. Lastly, the small local function \_loop() is called.





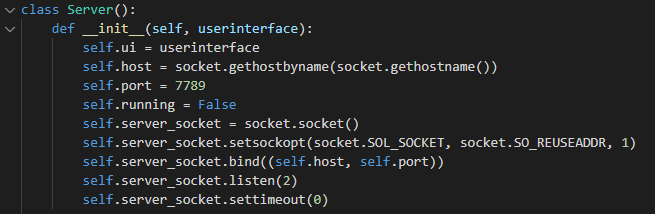
It simply starts the server object. The server class is handled by the interface class, as is the client class. Therein the existence of stop() also exits, to respectively stop the server running and set the running variable to false, terminating the thread.



onClick() is the function that calls submit() which submits the message written in an entry box to the open socket connection. The server demonstration works entirely exclusively from the packet sniffer, easily the program could grab the data running through it internally however that’s not packet capturing. The packets captured via the main program are all legitimately captured over the network, including packets of data sent across socket connections over ports, as seen by “self.client.client\_socket.send(usr\_message.encode())” which sends over the entered message in encoded format across the socket.

The userInterface() class also contains a few local functions that are for aesthetic purposes only, like button hover animations and button click animations.

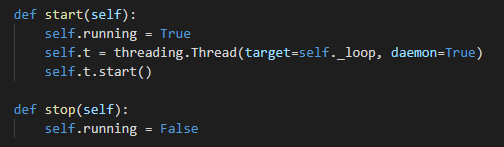
Server() creates the server side socket connection, it pairs up with the Client() class.



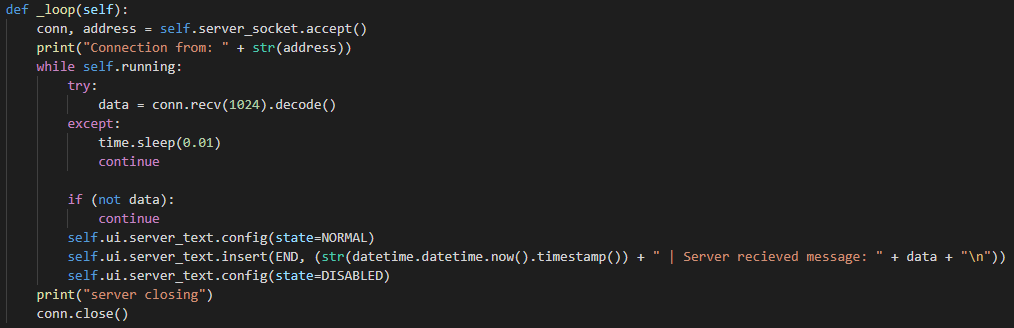
The constructor references the userInterface() class as a parent and then creates the socket parameters like host and port. I get the hostname by the hostname of the machine the program is running on, and the port is a custom port that is very very rarely in use over a network. The local running variable is set to false initially.

The important part of the constructor is the construction of the socket. It sets up the socket with certain parameters, socket.SO\_REUSEADDR means that the socket can rebind on the same port and ip address after being closed and reopened in one instance of the main program. socket.SOL\_SOCKET keeps any intervention of my code to a purely socket level as to not tamper with any data level socket transmissions.

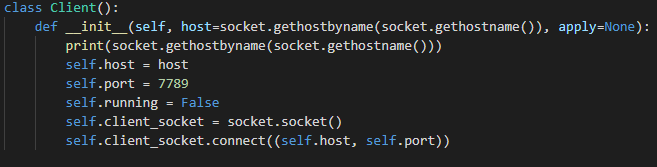
The constructor binds the socket and listens.



The server class is also sub-threaded from the userInterface class in the same file. It targets \_loop() and is in daemon mode, meaning it terminates when threads above it terminate. The stop function is used to set the flag and stop the thread to control termination.



\_loop() handles the data transmission on the server side, it accepts incoming connections and then accepts incoming data. It prints any data the socket receives to the userInterface server text box for the user. It receives split up messages in sizes under 1024 bytes repeatedly in a loop until an entire packet is sent, the socket then reassembles the split up bytes back into a packet.



The client is also initialised in the setup part of the server demonstration, and it’s a very basic client side socket connection that pairs up with the server. It uses the machine as the host as 7789 as a port then connects over a socket to the server side. I didn’t need to program an independent thread and flag check on the client because the client automatically ceases to function whenever the server does. It’s redundant to create two different thread flags for what is essentially one function.