A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1 Analysis

1.1 Identification and Background to the Problem

The problem I am trying to solve with my project is how to look at devices on a network from a "black box" perspective and gain information about what services are running etc. Services are programs which their entire purpose is to provide a *service* to other programs, for example a server hosting a website would be running a service whose purpose is to send the webpage to people who try to connect to the website.
There are many steps in-between a device turning on to interacting with the internet.

1. load networking drivers
2. Starting Dynamic Host Configuration Protocol (DHCP) daemon
3. Broadcasting DHCP request for an IP address
4. Get assigned an IP address
5. ???
6. Profit!!!

There are many more steps than I have listed above. Starting from a linux computer being switched on the first step is that the kernel needs to load the networking drivers. The kernel is the basis for the operating system, it is what interacts with the hardware in the most fundamental way. drivers are small bits of code which the kernel can load in order to interact with certain hardware modules such as the Network Interface Card (NIC) which is essential for interfacing with the network, hence the name.

Next once the kernel has loaded the required drivers and the system has booted the networking 'daemons' must be started. In linux a daemon is a program that runs all the time in the background to serve a specific purpose or utility. For example when I start my laptop the following daemons start upowerd (power management), systemd (manages the creation of all processes), dbus-daemon (manages inter-process communication), iwd (manages my WiFi connections) and finally Dynamic Host Configuration Protocol Client Daemon (DHCPCD) which manages all interactions with the network around DHCP.

Once the daemons are all started the DHCP client sends a discover message with the address 255.255.255.255 which is the IP limited broadcast address which means that whatever is listening at the other end will forward this packet on to everyone on the subnet. When the DHCP server on the subnet receives this message it reserves a free IP address for that client and then responds with a DHCP offer which contains the address the server is offering, the length of time the address is valid for and the subnet mask of the network. The client must then respond with a DHCP request message to request the offered address, this is in case of multiple DHCP servers offering addresses. Finally the DHCP server responds with a DHCP acknowledge message showing that it has received the request. Figure 1 shows a packet capture from my laptop where I turned WiFi off, started wireshark listening and plugged in an Ethernet cable, I have it showing only the DHCP packets so that it is clear to see the entire DHCP negotiation including the 255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column.

No.	Time	Source	Destination	Protocol	Info
6	0.983737378	0.0.0.0	255.255.255.255	DHCP	DHCP Discover
32	4.239092378	192.168.1.1	192.168.1.47	DHCP	DHCP Offer
34	4.239420587	0.0.0.0	255.255.255.255	DHCP	DHCP Request
36	4.241743101	192.168.1.1	192.168.1.47	DHCP	DHCP ACK

Figure 1: DHCP address negotiation

All computer networking is encapsulated in the Open Systems Interconnection model (OSI model) which has 7 layers:

7. Application: Applications Programming Interface (API)s, Hypertext transfer Protocol (HTTP), File Transfer Protocol (FTP) among others.
6. Presentation: encryption/decryption, encoding/decoding, decompression etc...
5. Session: Managing sessions, PHP Hypertext Processor (PHP) session IDs etc...
4. Transport: TCP and UDP among others.
3. Network: ICMP and IP among others.
2. Data Link: MAC addressing, Ethernet protocol etc...
1. Physical: The physical Ethernet cabling/NIC.

Each of these layers is essential to the running of the internet but a single communication might not include all of the layers.

These communications are all based on the most fundamental part of the internet: the packet. Packets are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information each specifying where the packet where should go next at a different level along with fundamentally the data/instructions contained in the innermost layer. When packets are sent between computers a certain number of layers are stripped off by each computer so that it knows where to send the packet next at which point it will add all the layers back again, this time with the instructions needed to go from the current computer to the next one on its route. Each of these layers actually consists of a number of fields at the start called a header some layers also append a footer to the end of the packet. The actual data being transferred in the packet can be quite literally anything, HTTP transfers websites so Hypertext Markup Language (HTML) files and images etc. . .

I'm going to use the example of getting a very simple static HTML page with an image inside. The code for the page is shown in listing 1. In figure 2 you can see how the page renders. However far more interestingly is how the browser retrieved the page, in figure 3 you can see the full sequence of packets that were exchanged for the browser to get the resources it needed to render the page. I am hosting the page using python3's http.server module which is super convenient and just makes the current directory open on port 8000 from there I can just navigate to /example.html and it will render the page. Breaking figure 3 down packet one shows the browser receiving the request from the user to display `http://192.168.1.47:8000/example.html` and attempting to connect to 192.168.1.47 on port 8000. Packets two and three show the negotiation of this request through to the full connection being made. The browser now makes an HTTP GET request for the page example.html over the established TCP connection as shown in packet 4. The server then acknowledges the request and sends a packet with the PSH flag set as shown in packets 6 and 7. The PSH flag is a request to the browser to say that it is OK to received the buffered data, i.e. example.html. The browser then sends back an acknowledgement and the server sends the page as shown in packets 7 and 8. Finally the browser sends a final acknowledgement of having received the page before initiating a graceful session teardown by sending a FIN ACK packet which indicates the end of a session. Once the server responds to the FIN ACK with it's own the browser sends a final acknowledgement. This then repeats itself when the browser parses the HTML and realises theres an image which it needs to get from the server as well, except the image is a larger file and so takes a few more PSH packets.

This shows clearly the interaction between each of the different layers in the OSI model, the browser at level 7: Application rendering the webpage. Level 6: Presentation is skipped as we have no files which need to be served compressed because they are so large. Level 5: Session is shown by the TCP session negotiation and graceful teardown of the TCP session. Level 4: Transport is shown when the image and webpage are transferred from the server to the browser. Level 3/2/1 are shown in figure 5 where you can see the IP layer information along with Ethernet II and finally frame 4 which is the bytes that went down the wire.

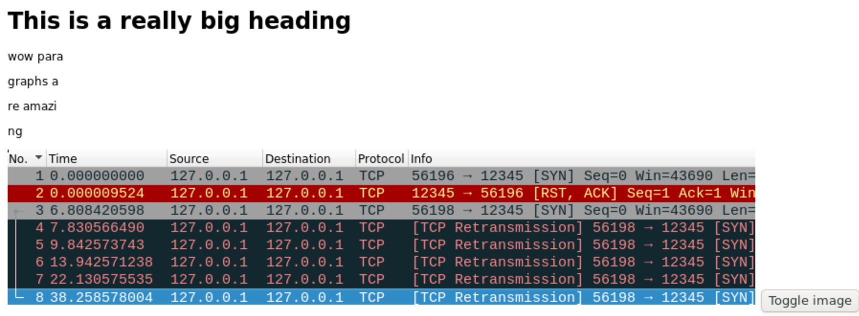

Figure 2: A basic static HTML webpage.

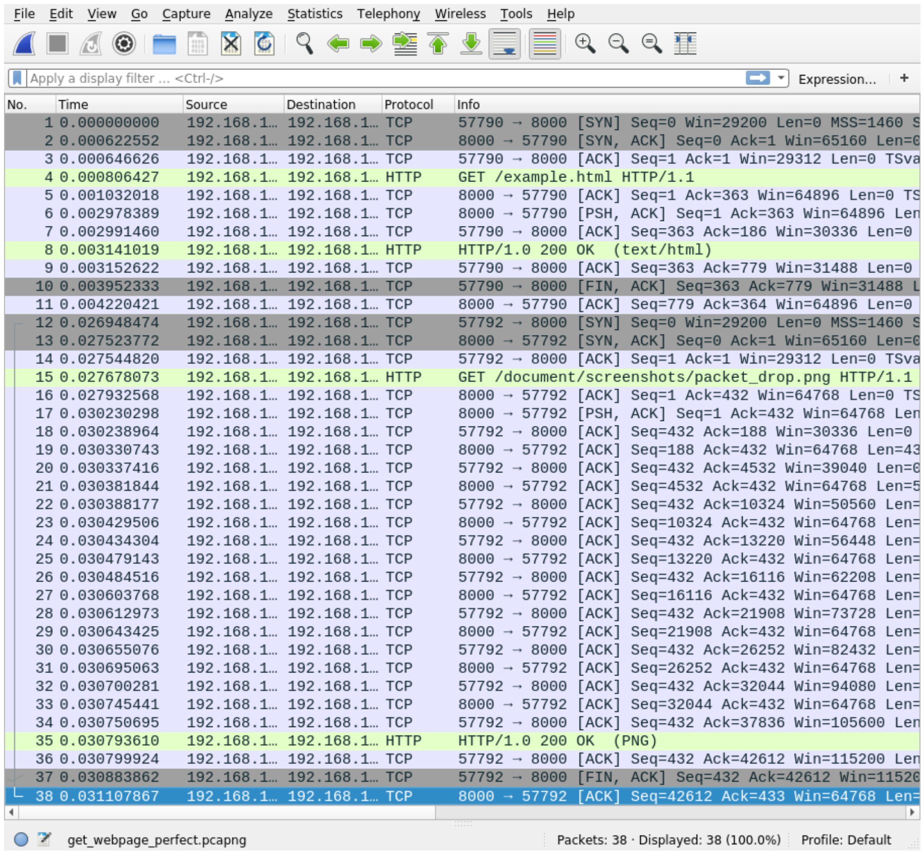

Figure 3: A full chain of packets that shows retrieving a basic webpage from the server.

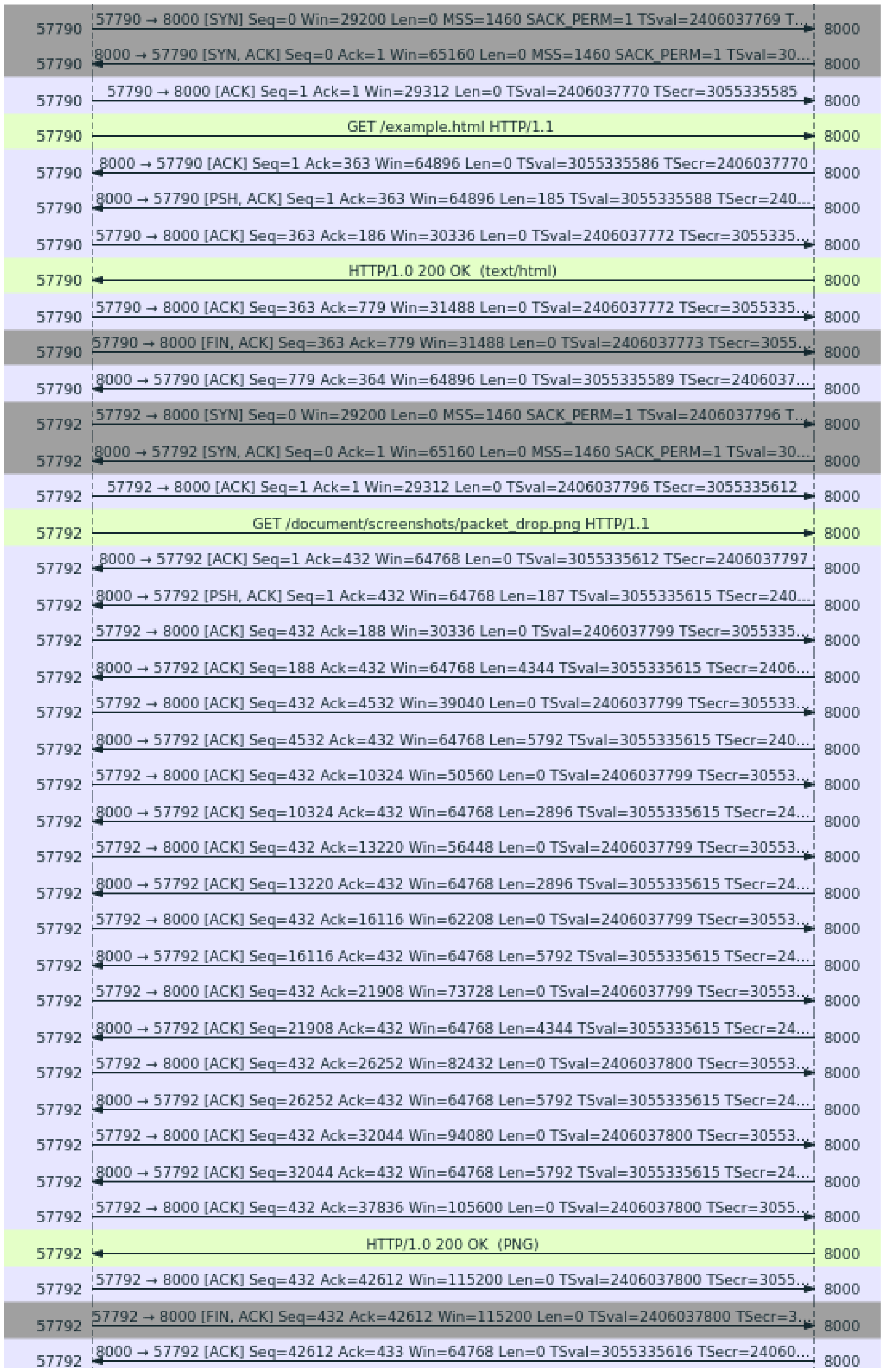

Figure 4: Ladder diagram of figure 3.

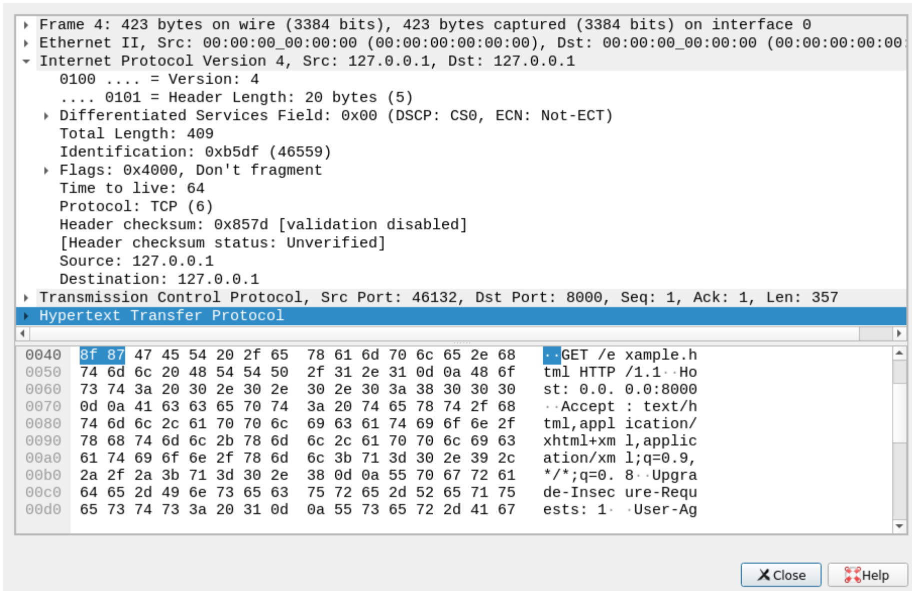

Figure 5: A look inside a TCP packet.

Listing 1: example.html

```
 <!DOCTYPE html>
<html>
<head>
<title>Wow I can add titles</title>
</head>
<body>

<h1>This is a really big heading</h1>
<p>wow para</p>
<p>graphs a</p>
<p>re amazi</p>
<p>ng</p>
  <script type="text/javascript">
    function imgtog() {
      if (document.getElementById("img").style.display == "none") {
        document.getElementById("img").style = "block"
      } else {
        document.getElementById("img").style.display = "none"
      }
    }

  </script>

```

```
<img id="img" src="document/screenshots/packet_drop.png">

<button onclick="imgtog()">Toggle image</button>


</body>
</html>
```

1.2 Analysis of problem

The problem with looking at a network from the outside is that the purpose of the network is to allow communication inside of the network, thus very little is exposed externally. This presents a challenge as we want to know what is on the network as well as what each of them is running which is not always possible due to the limited information that services will reveal about themselves. Firewalls also play large part in making scanning networks difficult as sometimes they simply drop packets instead of sending a Transmission Control Protocol (TCP) RST packet (reset connection packet). When firewalls drop packets it becomes exponentially more difficult as you don't know whether your packet was corrupted or lost in transit or if it was just dropped.

To demonstrate this I will show three things:

1. A successful connection over TCP.
2. An attempted connection to a closed port.
3. An attempted connection with a firewall rule to drop packets.

Firstly A successful TCP connection. For a TCP connection to be established there is a three way handshake between the communicating machines. Firstly the machine trying to establish the connection sends a TCP SYN packet to the other machine, this packet holds a dual purpose, to ask for a connection and if it is accepted to SYNchronise the sequence numbers being used to detect whether packets have been lost in transport. The receiving machine then replies with a TCP SYN ACK which confirms the starting sequence number with the SYN part and ACKnowledges the connection request. The sending machine then acknowledges this by sending a final TCP ACK packet back. This connection initialisation is shown in figure 6 by packets one, two and three. Data transfer can then commence by sending a TCP packet with the PSH and ACK flags set along with the data in the data portion of the packet, this is shown in figure 9 where wireshark allows us to take a look inside the packet to see the data being sent in the packet along with the PSH and ACK flags being set. The code I used to generate these is shown in figures 7 and 8. Breaking the code down in figure 8 you can see me initialising a socket object then I bind it to localhost (127.0.0.1) port 12345 localhost is just an address which allows connections between programs running on the same computer as connections are looped back onto the current machine, hence its alternative name: the loopback address. I then tell it to listen for incoming connections, the one just means how many connections to keep as a backlog. I then accept the connection from the program in figure 7, line 3. I then tell the program to listen for up to 1024 bytes in the data part of any TCP packets sent. The program in figure 7 then sends some data which we then see printed to the screen in figure 8, both programs then close the connection.

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [SYN] Seq=0
2	0.000019294	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [SYN, ACK]
3	0.000033431	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=1
4	53.378941809	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [PSH, ACK]
5	53.378958066	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=1
6	65.928944995	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [FIN, ACK]
7	65.936113471	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=3
8	85.536923935	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [FIN, ACK]
9	85.536940026	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=2

Figure 6: Packets starting a TCP session, transferring some data then ending it.

```
In [1]: import socket

In [2]: sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: sender.connect(("127.0.0.1", 12345))

In [4]: sender.send(b"hi I'm data what's your name? "*10)
Out[4]: 300

In [5]: sender.close()
```

Figure 7: Transferring some basic text data over a TCP connection.

```
In [1]: import socket

In [2]: receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: receiver.bind(("127.0.0.1", 12345))

In [4]: receiver.listen(1)

In [5]: connection, address = receiver.accept()

In [6]: connection.recv(1024)
Out[6]: b"hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm
 data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's you
r name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? "

In [7]: connection.close()
```

Figure 8: Receiving some basic text data over a TCP connection.

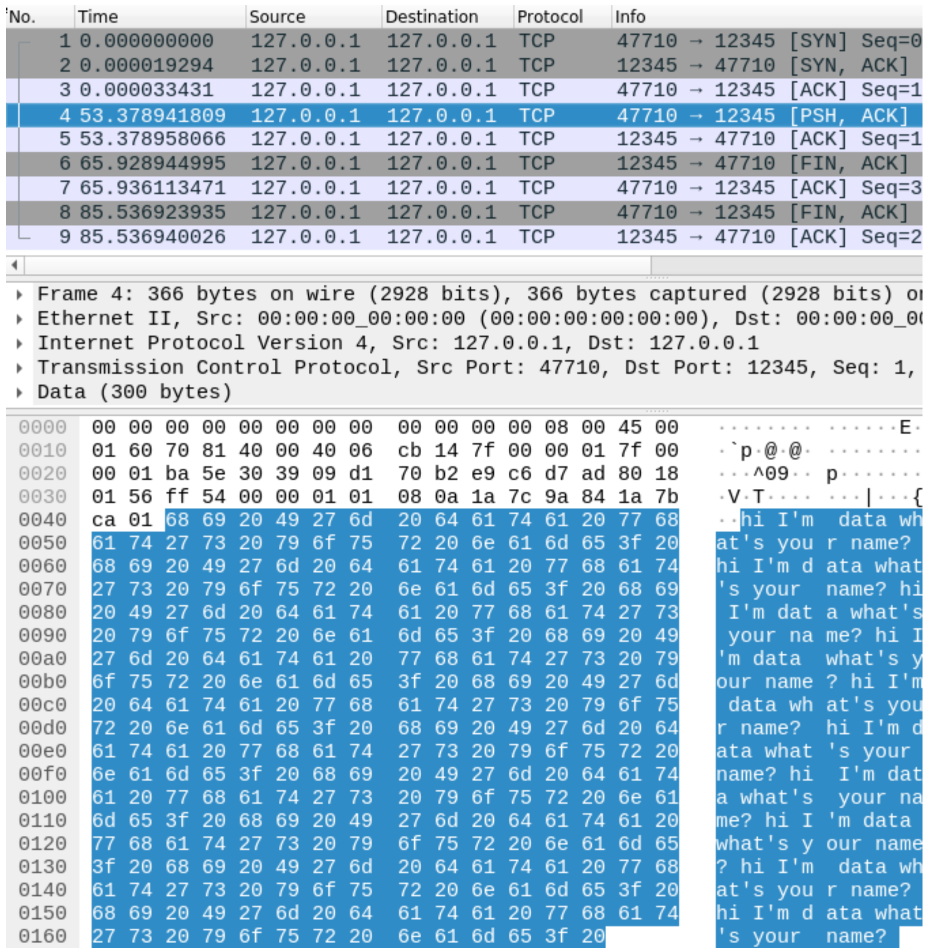

Figure 9: Highlighted packet carrying the data being transferred in figure 7.

Next an attempted connection to a closed port. In figure 10 packet one you can see the same TCP SYN packet as we saw in the attempted connection to an open port, as you would expect. The difference comes in the next packet with the TCP RST flag being sent back. This flag means to reset the connection, or if the connection is not yet established as in this case it means that the port is closed, hence why the packet is highlighted red in figure 10. The code used to generate this is shown in figure 11 line two shows the initialisation of a socket object. In line 3 the program tries to connect to port 12345 on localhost again, except this time we get a connection refused error back this shows us that the remote host sent a TCP RST packet back, which is reflected in figure 10.

Finally I will show a connection where the firewall is configured to drop the packet. However first I will explain a bit about firewalls and how they work.

Firewalls are essentially the gatekeepers of the internet they decide whether a packet gets to pass or whether they shall not pass. Firewalls work by a set of rules which decide what happens to it. A rule might be that it is coming from a certain IP address or has a certain destination port. The actions taken after the packet has had it's fate decided by the rules can be one of the following three (on iptables on linux): ACCEPT, DROP and RETURN, accept does exactly what you think it would an lets the packet through, drop quite literally just drops the packet and sends no reply whatsoever, return is more complicated and has no effect on how port scanning is done and as such we will ignore it. A common set of rules for something like a webserver would be to DROP all incoming packets and then allow exceptions for certain ports i.e. port 80 for HTTP or 443 for Hypertext transfer Protocol Secure (HTTPS). I will be using a linux utility called iptables for implementing all firewall rules on my system for demonstration purposes. Packet number three in figure 10 shows the connection request from line 4 of 11 except that I have enabled a firewall rule to drop all packets from the address 127.0.0.1, using the iptables command as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. This command reads as for all packets arriving (`-I INPUT`) with source address 127.0.0.1 (`-s 127.0.0.1`) drop them sending no response (`-j DROP`). With this firewall rule in place you can see in figure 10 packet 3 receives no response and as such python assumes that the packet just got lost and as such tries to send the packet again repeatedly, this continued for more than 30 seconds before a stopped it as shown by the time column in figure 10 and the final `KeyboardInterrupt` in figure 11. The amount of time that a system will wait still trying to reconnect depends on the OS and a other factors but the minimum time is 100 seconds as specified by RFC 1122, on most systems it will be between 13 and 30 minutes according the linux manual page on TCP.

```
man 7 tcp:
tcp_retries2 (integer; default: 15; since Linux 2.2)
  The maximum number of times a TCP packet is retransmitted in
  established state before giving up. The default value is 15,
  which corresponds to a duration of approximately between 13 to 30
  minutes, depending on the retransmission timeout. The RFC 1122
  specified minimum limit of 100 seconds is typically deemed too short.
```

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	56196 → 12345 [SYN] Seq=0 Win=43690 Len=
2	0.000009524	127.0.0.1	127.0.0.1	TCP	12345 → 56196 [RST, ACK] Seq=1 Ack=1 Win
3	6.808420598	127.0.0.1	127.0.0.1	TCP	56198 → 12345 [SYN] Seq=0 Win=43690 Len=
4	7.830566490	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
5	9.842573743	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
6	13.942571238	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
7	22.130575535	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
8	38.258578004	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]

Figure 10: Attempted connection to a closed port with and without firewall rule to drop packets.

```
In [1]: import socket

In [2]: a = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: a.connect(("127.0.0.1", 12345))
---------------------------------------------------------------------------
ConnectionRefusedError                    Traceback (most recent call last)
<ipython-input-3-fbc96d60b5f2> in <module>
----> 1 a.connect(("127.0.0.1", 12345))

ConnectionRefusedError: [Errno 111] Connection refused

In [4]: a.connect(("127.0.0.1", 12345))
^C---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
```

Figure 11: The code used to produce firewall packet dropping example in figure 10

1.3 Numbered List of Objectives (also called Success Criteria, the end user requirements)

1. Show a basic usage message when called with no arguments.
2. Show a help message when called with `-help` or `-h`.
3. Scan the 1000 most commonly used TCP ports when called with just an IP address.
4. Scan the ports specified by `-p <ports>` or `-ports <ports>`.
5. Parse either a comma separated list of ports e.g. `1,2,3,4` or a range specified set of ports e.g. `1-4`.
6. Scan all ports in each scan type when called with `-p-`.
7. Don't scan the ports specified by `-exclude-ports <ports>` in any scan.
8. Expand a Classless Inter-Domain Routing (CIDR) specified subnet when used in the target specification.
9. List all of the IP addresses that would be scanned when given the `-sL` flag.
10. Only ping each specified address when supplied the `-sn` flag.
11. When doing a ping scan (`-sn`) display the Time To Live (TTL) and latency of each host.
12. Don't ping each of the hosts before scanning to check if they are up when supplied with the `-Pn` flag.

13. Perform a TCPSYN scan on the 1000 most common TCP ports on each target specified when given the `-sS` flag.
14. Perform a TCP `Connect()` scan on the 1000 most common TCP ports on each target specified when given the `-sT` flag.
15. Perform aUser Datagram Protocol (UDP)scan on the 1000 most commonUDP ports when on each target specified when given the `-sU` flag.
16. Perform version detection on the services running on each of the hosts specific when given the `-sV` flag.

1.4 Description of current system or existing solutions

Nmap is currently the most popular tool for doingportscanning and host enumeration. It supports the following scanning types:

- TCP: SYN
- TCP: `Connect()`
- TCP: ACK
- TCP: Window
- TCP: Maimon
- TCP: Null
- TCP: FIN
- TCP: Xmas
- UDP
- Zombie host/idle
- Stream Control Transmission Protocol (SCTP): INIT
- SCTP: COOKIE-ECHO
- IP protocol scan
- FTP: bounce scan

As well as supporting a vast array of scanning types it also can do service version detection and operating system detection via custom probes. Nmap also has script scanning which allows the user to write a script specifying exactly how they want to scan e.g. to circumvent port knocking (where packets must be sent to a sequence of ports in order before access to the finalportis allowed). It also supports a plethora of options to avoid firewalls or Intrusion Detection System (IDS) such as sending packets with spoofed checksums/source addresses and sending decoy probes. Nmap can do many more things than I have listed above as is illustrated quite clearly by the fact there is an entire working on using nmap (https://nmap.org/book/)

1.5 Prospective Users

The prospective users of this system would be system administrators, penetration testers or network engineers. In my case my prospective users would be my school's system administrators and it would allow them to see an outsiders perspective on for example the server running the school's website page or to see if any of the programs on the servers were leaking information through banners etc. (most services send a banner with information like what protocol version they use and other information)

1.6 Data Dictionary

I looked this up and it seemed to be related to database management systems.

https://en.wikipedia.org/wiki/Data_dictionary

1.7 Data Flow Diagram

This seems to be fairly relevant and to do with how data goes through my program i.e. going from the network to my port scanner into a target object and other scanners before version detection and finally displaying to the user. Make a flowchart for this.

https://en.wikipedia.org/wiki/Data-flow_diagram

1.8 Data Sources

Not really sure about this.

1.9 Description of Solution Details, OOP/Mobile/Networking

To do all forms of scanning other than `Connect()` scanning and version detection, custom packets are made to allow the half open scanning (no full connection is made to the host) scanning used inTCPSYN scanning. Making custom packets is quite difficult because the endianness (the order the bytes are interpreted in: big endian, most significant byte first, little endian, least significant byte first) affects how all the information packed into the packet is interpreted by the network switch, for example theIP address192.168.1.58 packed in big endian form but interpreted being in little endian form comes out as 58.1.168.192 which is a completely different address and will mean the packet is not routed to the correct host. As well as the issues with byte order and the interpretation of information at different points the checksum which is embedded into the packet is calculated from a psuedo-header calculated from information in the underlying IP header and all of this has to be calculated in the right byte order (endianness).

I have used Python's multiprocessing module to allow me to spawn another process which listens for responses from hosts and waits for a certain amount of time before returning information on what hosts responded and in the case of ping scanning also metadata about how they responded.

In version detection scanning the relationship between the data sources and modules used is quite complex so I have used an Object Oriented Programming (OOP) approach to group the methods that act on the data along with the data itself. For example each probe defined in the `nmap-service-probes` file can be sent to a host and matched against a list of match directives stored in the probe, the probe class has a scan method which sends it's probe to the host and then automatically runs match and soft-match directives against the information returned by the probe.

Parsing the match and softmatch directives was quite difficult because they include regular expressions with special characters such as newlines and carriage returns in the form of `\n` and `\r` characters which python escapes to `\\n` and `\\r`. Which instead of matching a newline character and a carriage return will match a literal backslash and then an n or an r which is not what we want. To fix this I have to substitute newline and carriage returns back in where I find `\\r` and `\\r`.

1.10 Acceptable Limitations (Supplementary)

Originally I had planned to include dedicated operating system detection as an option however I ran out of time having implemented version detection. However it still does Operating system detection partially as some services are linux only and while doing service and version detection especially the Common Platform Enumeration (CPE) parts of the matched service/version will contain operating system information, such as microsoft ActiveSync would indicate that the system being scanned was a windows system which is reflected in the match directive and attached CPE information: `match activesync m|^.\0\x01\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0.*\0\0\0$|s p/Microsoft ActiveSync/ o/Windows/ cpe:/a:microsoft:activesync/ cpe:/o:microsoft:windows/a`

1.11 Data Volumes (Supplementary)

This seems to be about the volume of data stored in a database. https://stackoverflow.com/questions/5566841/what-are-data-volumes#5567390

1.12 Test Strategy

I am going to use two different methods to test my program:

1. Unit testing
2. Wireshark

I am using two separate testing strategies because they are both good at different things, both of which I need to show that my project works. Firstly I am using unit testing to test some general purpose functions which are pure functions (are independent of the current state of the machine) such as `ip_range()` and other functions which I can just check the returned value against what it should be.

Wireshark is useful for the other half of the program which uses impure functions and the low level networking e.g. `make_tcp_packet()`. Wireshark makes this easy by allowing capture of all the packets going over the wire, as well as this it has a vast array of packet decoders (2231 in my install) which it can use to dissect almost any packet that would be on the network. The main benefit of wireshark is that I can see my scanners sending packets and then check whether the parsers that I have written for the different protocols are working. I can also check that the checksums in each of the various protocols is valid as wireshark does checksum verification for various protocols.

2 Design

2.1 Overall System Design (High Level Overview)

There are two types of scanning implemented for different scan types in my program.

- `Connect()`
- version
- listener / sender

`Connect()` scanning is the simplest in that it takes in a list of ports and simply calls the `socket.connect()` method on it and sees whether it can connect or not and the ports are marked accordingly as open or closed.

Version scanning is very similar to `Connect()` scanning in that it takes in a list of ports and connects to them, except it then sends a probe to the target to elicit a response and gain some information about the service running behind the port.

Listener / sender scanning does exactly what it says on the tin: it sets up a "listener" in another process to listen for responses from the host which the "sender" is sending packets to. It can then differentiate between open, open|filtered, filtered and closed ports based on whether it receives a packet back and what flags (part ofTCP packets are a one byte long section which store "flags" where each bit in the byte represents a different flag) are set in the received packet.

2.2 Design of User Interfaces HCI

I have designed my system to have a similar interface to the most common tool currently used: nmap this is because I believe that having a familiar interface will not only make it easier for someone who is familiar with nmap to use my tool it also makes it so that anything learnt using either tool is applicable to both which benefits everyone.

Based on this perception I have used the same option flags as nmap as well as similar help messages and an identical call signature (how the program is used on the command line). Running `./netscan.py <options> <target_spec>` is identical to `nmap <options> <target_spec>` in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below you can see the help message generated by `./netscan.py --help`.

```
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

positional arguments:
  target_spec           specify what to scan, i.e. 192.168.1.0/24

optional arguments:
  -h, --help            show this help message and exit
  -Pn                   assume hosts are up
  -sL                   list targets
  -sn                   disable port scanning
  -sS                   TCP SYN scan
  -sT                   TCP connect scan
  -sU                   UDP scan
  -sV                   version scan
  -p PORTS, --ports PORTS
                        scan specified ports
  --exclude_ports EXCLUDE_PORTS
                        ports to exclude from the scan
```

It shows clearly which are required arguments and which are optional ones, as well as what each argument actually does. It also allows some some arguments to be called with either a short format e.g. `-p` and with a most verbose format `--ports` this allows the user to be clearer if they are using the tool as part of an automated script to perform scanning as it is more immediately obvious what the more verbose flags do.

2.3 System Algorithms (Flowcharts)

When I have finished the first draft of the text bits I will add pictures / flowcharts

2.4 Input data Validation

My program takes very little input from the user which means that there is a very low chance of the program crashing due to user input error as the errors are detected All data which is entered is either parsed using a regular expression with the case of the ports directive (`-p`) or is run through checking functions like `ip_utils.is_valid_ip`. As well as using these checking functions whenever anIP addressis converted between “long form” and “dot form” which is used in every type of scanning.

2.5 Proposed Algorithms for complex structures (flow charts or Pseudo Code)

Algorithm 1 My algorithm for turning a CIDR specified subnet into a list of actual IP addresses

procedure IP_RANGE
    network_bits $\leftarrow$ number of network bits specified
    ip $\leftarrow$ base IP address
    mask $\leftarrow 0$
    for $maskbit \leftarrow (32 - network\_bits), 31$ **do**
        $mask \leftarrow mask + 2^{maskbit}$
    lower_bound $\leftarrow$ *ip* AND *mask*   ▷ zero the last 32-*network_bits*
    upper_bound $\leftarrow$ *ip* OR (*mask* XOR 0xFFFFFFFF)   ▷ turn the last 32-*network_bits* to ones
    addresses $\leftarrow$ empty list
    for *address* $\leftarrow$ *lower_bound*, *upper_bound* **do**
        append CONVERT_TO_DOT(*address*) to *addresses*
    return *addresses*

Algorithm 2 My algorithm for pretty-printing a dictionary of lists ofportnumbers such that ranges are specified as start-end instead of start,start+1,...,end

```
procedure COLLAPSE
    port_dictionary ← dictionary of lists ofportnumbers
    key_results ← empty list        ▷ stores the formatted result for each key
    for key in port_dictionary do
        ports ← port_dict[key]
        result ← key + ":{"
        if ports is empty then
            new_sequence ← FALSE
            for index ← 1, (length of ports) − 1 do
                port = ports[index]
                if index = 0 then
                    result ← result + ports[0]        ▷ append the first element
                    if ports[index+1] = port + 1 then
                        result ← result + "-"        ▷ begin a new sequence
                    else
                        result ← result + ","        ▷ not a sequence
                else if port + 1 ≠ ports[index+1] then        ▷ break in sequence
                    result ← result + port + ","
                    new_sequence ← TRUE
                else if port + 1 = ports[index+1] & new_sequence then
                    result ← result + "-"
                    new_sequence ← FALSE
            result ← result + ports[(length of ports)-1] + "}"
            append result to key_results
    return "{" + (key_results separated by ", ") + "}"
```

2.6 Design Data Dictionary

I have no idea what this means. All I can find is that it relates to database structure???

3 Technical Solution

3.1 Program Listing

3.2 Comments (Core)

3.3 Overview to direct the examiner to areas of complexity and explain design evidence

4 Testing

4.1 Test Plan

4.2 Test Table / Testing Evidence (Core: lots of screenshots)

5 Evaluation

5.1 Reflection on final outcome

5.2 Evaluation against objectives, end user feedback

5.3 Potential improvements

6 Appendices

You may show you program listing here

User feedback and survey data

Glossary

API Applications Programming Interface 3

banner A short piece of text which a service with send to identify itself when it receives a connection request. Often contains information such as version number etc... 16

black box Looking at something from an outsider's perspective knowing nothing about how it works internally. 2

checksum A checksum is a value calculated from a mathematical algorithm which is sent with the packet to its destination to allow the recipient to check whether the packet was corrupted on the way. 15, 16, 18

CIDR Classless Inter-Domain Routing 14, 20

CPE Common Platform Enumeration 17

daemon A process that runs forever in the background to facilitate other programs. 2

dbus-daemon A daemon which enable a common interface for inter-process communication. 2

DHCP Dynamic Host Configuration Protocol 2, 3

DHCPCD Dynamic Host Configuration Protocol Client Daemon 2

driver A tiny software module which is loaded into the kernel when the computer boots up, They mainly interface with hardware and are often very specific for each piece of hardware. 2

FTP File Transfer Protocol 3, 15

half open scanning Half open scanning is where no full connection to the host is made, only one to solicit a response and then once that response is received no further packets are sent, leaving the connection "half open". 16

header A header is the first few bytes at the start of a packet often consisting of information on where to send the packet next, can also contain information though. 3, 16

HTML Hypertext Markup Language 4, 5

HTTP Hypertext transfer Protocol 3, 4, 13

HTTPS Hypertext transfer Protocol Secure 13

IDS Intrusion Detection System 15

IP address Every computer on a network has a unique IP address assigned to them, which is used to identify where exactly message sent by computers are meant to go. 2, 13, 14, 16, 20

kernel The kernel is the foundation of an operating system and it serves as the main interface between the software running on the system and the underlying hardware it performs task such as processor scheduling and managing input/output operations. 2

NIC Network Interface Card 2, 3

OOP Object Oriented Programming 17

OSI model Open Systems Interconnection model 3

packet Packets are simply a list of bytes which contains packed values such as to and from address and they are the basis for almost all inter-computer communications. 2, 3, 4, 6, 8, 9, 13, 14, 15, 16, 18

PHP PHP Hypertext Processor 3

port Computers have "ports" for each protocol which can be connected to separately, this makes up part of a "socket" connection. 14, 15, 18, 19, 20, 21

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 15

SCTP Stream Control Transmission Protocol 15

server A server is any computer which it's purpose is to provide resources to others, either humans or other computers for purposes from hosting website or just as a resource of large computational power. 2, 16

service A service is something running on a machine that offers a service to either other programs on the computer or to people on the internet. 2, 9, 15, 16, 17, 18

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 2, 14, 20

systemd A daemon for controlling what is run when the system starts. 2

TCP Transmission Control Protocol 9, 12, 13, 14, 15, 16, 18

UDP User Datagram Protocol 15

upowerd Manages the power supplied to the system: charging, battery usage etc. . . 2