A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

# Analysis

## Identification and Background to the Problem

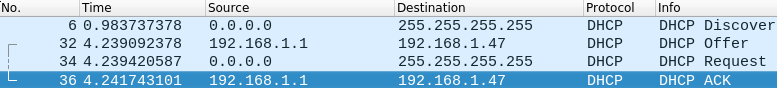
The problem I am trying to solve with my project is how to look at devices on a network from a “bbox” perspective and gain information about what services are running etc. Services are programs which their entire purpose is to provide a *service* to other programs, for example a server hosting a website 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 dhcp daemon
3. Broadcasting dhcp request for an ipaddr
4. Get assigned an ipaddr

There are many more steps than I have listed above but these are the most important ones. 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 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 upow (power management), sysd (manages the creation of all processes), dbus (manages inter-process communication), iwd (manages my WiFi connections) and finally dhcpcd which manages all interactions with the network around dhcp.

Once the daemons are all started the dhcp client can now take issue commands to the daemon for it to carry out. The dhcp client is simply a daemon that runs in the background to carry out any interactions between the current machine and the dhcp server. The dhcp server is normally the WiFi router or network switch for the local network and it manages a list of which computer has which ipaddr and negotiates with new computers trying to join a network to get them a free ipaddr. The dhcp client starts the dhcp address negotiation with the server by sending 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 pkt on to everyone on the subnet. When the dhcp server (normally the router, sometimes a separate machine) on the subnet receives this message it reserves a free ipaddr 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 [[dhcp\_negotiate]](#dhcp_negotiate) shows a pkt 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 pkts 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. I mention using wireshark to do packet capturing above without explaining what either packet capturing or wireshark are so I will do that here. Packets I define below and wireshark is simply a tool which intercepts all the network communications on a single computer and records them to a file as well as displaying them to the user as well as performing some analysis and dissecting each of the protocols used. This means that I can record the dhcp negotiation shown below and show it to you using wireshark to get all the information out of the packets being sent over the wire.



dhcp address negotiation

[dhcp\_negotiate]

All computer networking is encapsulated in the osi which has 7 layers:

Application: apis, http, ftp among others.

Presentation: encryption/decryption, encoding/decoding, decompression etc…

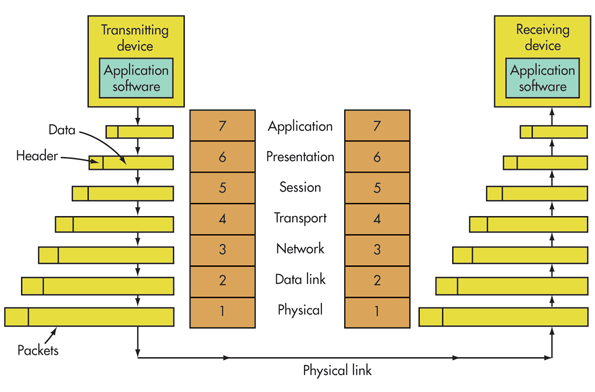
Session: Managing sessions, php session IDs etc…

Transport: TCP and UDP among others.

Network: ICMP and IP among others.

Data Link: MAC addressing, Ethernet protocol etc…

Physical: The physical Ethernet cabling/nic.



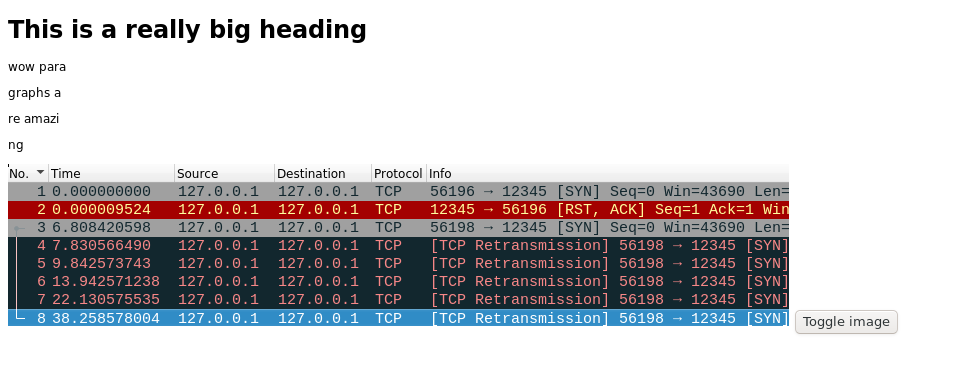
OSI model diagram, source: https://www.electronicdesign.com

[osi\_model]

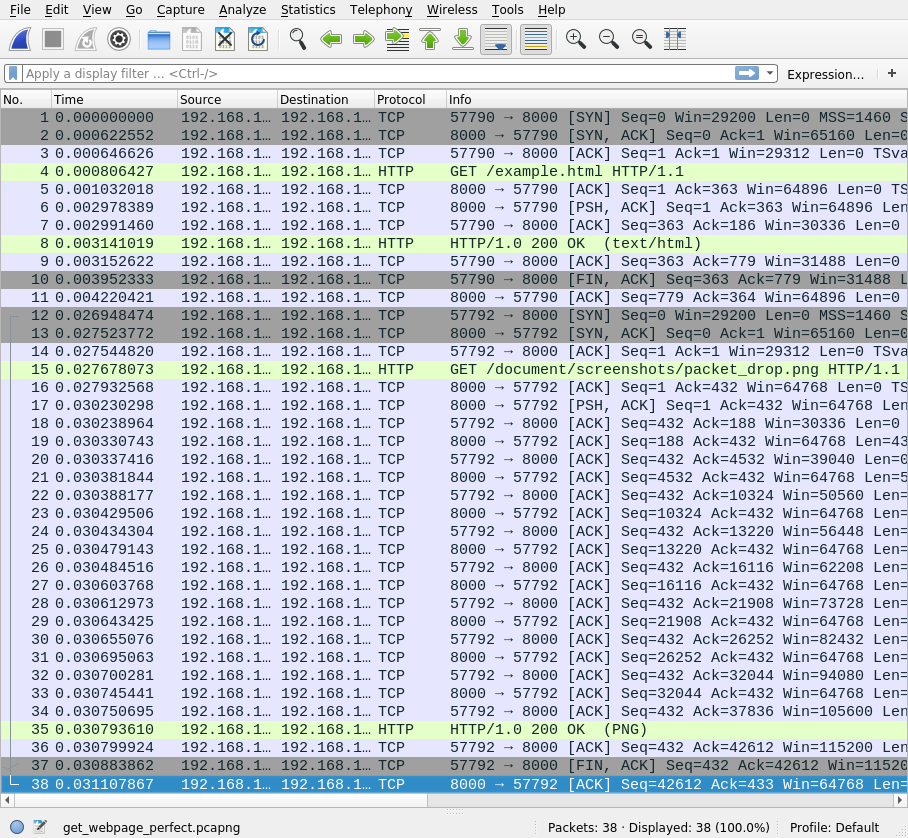
Each of these layers is essential to the running of the internet but a single communication might not include all of the layers. These communications are all based on the most fundamental part of the internet: the pkt. Packets are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information each specifying where the pkt where should go next at a different level along with fundamentally the data/instructions contained in the innermost layer. When pkts are sent between computers a certain number of layers are stripped off by each computer so that it knows where to send the pkt 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 html files and images etc…. In particular there are two pieces of information stored in headers which together define the final destination of the packet: the ipaddr and the port number. The ipaddr defines the destination machine and the port number defines which “port” on the remote machine the packet should be sent to. Ports are essential entrances to a computer, for example if a computer was a hotel the ipaddr would be the address and location of the hotel and the port number would be the room inside the hotel. There are 65535 ports and 0 is a special reserved port. Both tcp and udp use ports, tcp ports are mainly used for transferring data where reliability is a concern, as tcp has built in checks for packet loss whereas udp does not and as such is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing games.

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 [[examplepage]](#examplepage). In figure [[basicwebpage]](#basicwebpage) you can see how the page renders. However far more interestingly is how the browser retrieved the page, in figure [[getrequest]](#getrequest) you can see the full sequence of pkts 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 [[getrequest]](#getrequest) down pkt 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 pkt 4. The server then acknowledges the request and sends a pkt with the PSH flag set as shown in pkts 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 pkts 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 pkt 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 pkts.

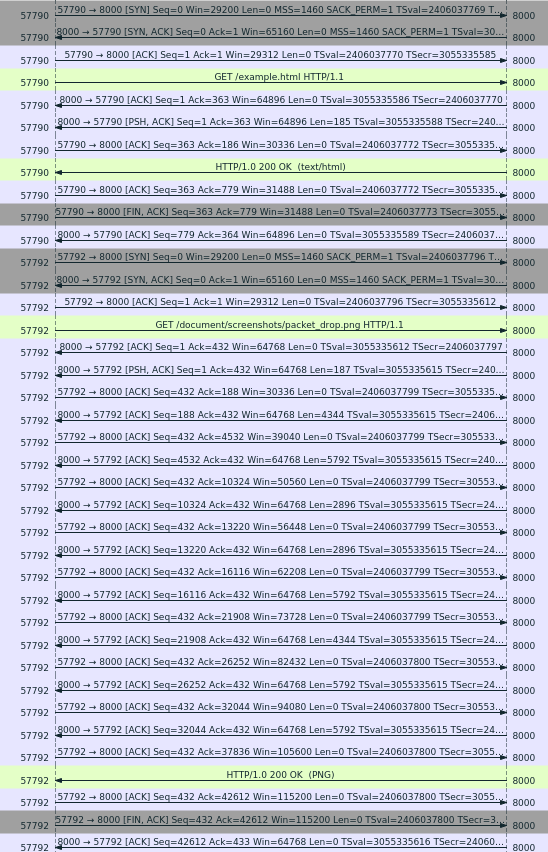
This shows clearly the interaction between each of the different layers in the OSI model, the browser at level 7: Application rendering the webpage. Level 6: Presentation is skipped as we have no files which need to be served compressed because they are so large. Level 5: Session is shown by the TCP 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 [[deconstructed]](#deconstructed) 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.



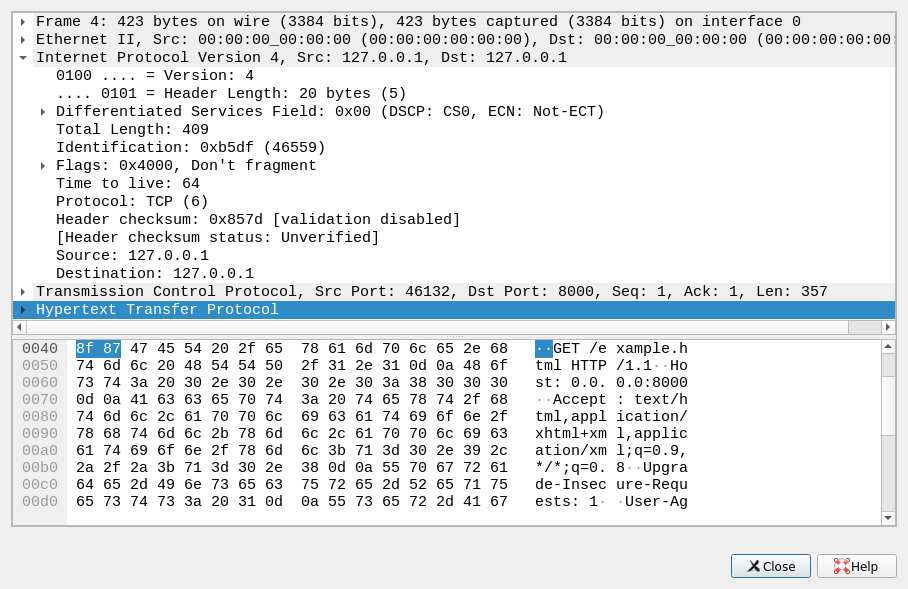
A basic static html webpage.



A full chain of pkts that shows retrieving a basic webpage from the server.



Ladder diagram of figure [[getrequest]](#getrequest).



A look inside a TCP pkt.

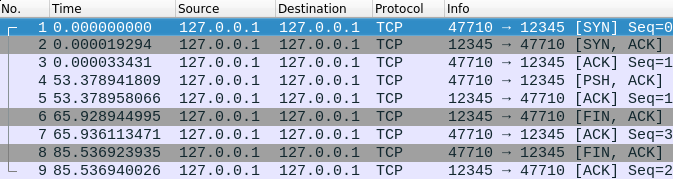
<!DOCTYPE html>  
<html>  
<head>  
<title>Wow I can add titles</title>  
</head>  
<body>  
  
<h1>This is a really big heading</h1>  
<p>wow para</p>  
<p>graphs a</p>  
<p>re amazi</p>  
<p>ng</p>  
 <script type="text/javascript">  
 function imgtog() {  
 if (document.getElementById("img").style.display == "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>

## 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 pkts instead of sending a tcp RST pkt (reset connection pkt). When firewalls drop pkts it becomes exponentially more difficult as you don’t know whether your pkt was corrupted or lost in transit or if it was just dropped.  
  
To demonstrate this I will show three things:

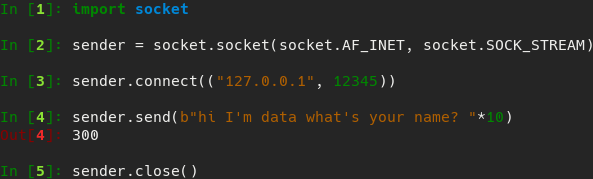
1. A successful connection over tcp.
2. An attempted connection to a closed port.
3. An attempted connection with a firewall rule to drop packets.

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 [[data\_transfer]](#data_transfer) 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 [[data]](#data) 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 [[sender]](#sender) and [[receiver]](#receiver). Breaking the code down in figure [[receiver]](#receiver) 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 [[sender]](#sender), 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 [[sender]](#sender) then sends some data which we then see printed to the screen in figure [[receiver]](#receiver), both programs then close the connection.

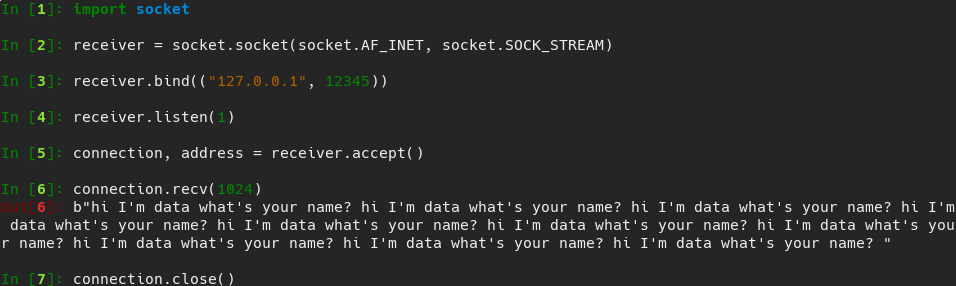


Packets starting a TCP session, transferring some data then ending it.

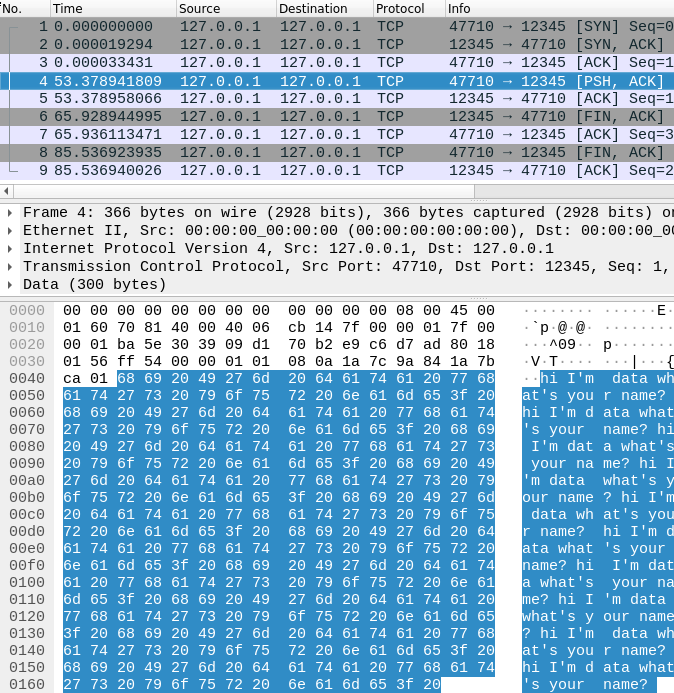
[data\_transfer]



Transferring some basic text data over a TCP connection.



Receiving some basic text data over a TCP connection.

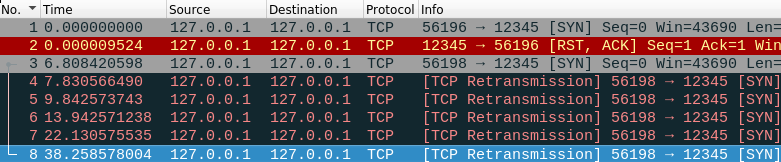


Highlighted packet carrying the data being transferred in figure [[sender]](#sender).

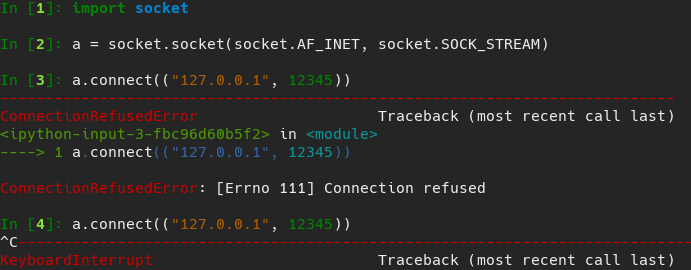
Next an attempted connection to a closed port. In figure [[firewall]](#firewall) packet one you can see the same tcp SYN packet as we saw in the attempted connection to an open port, as you would expect. The difference comes in the next packet with the tcp RST flag being sent back. This flag means to 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 [[firewall]](#firewall). The code used to generate this is shown in figure [[firewall\_code]](#firewall_code) 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 [[firewall]](#firewall).

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 ipaddr 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 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 [[firewall]](#firewall) shows the connection request from line 4 of [[firewall\_code]](#firewall_code) except that I have enabled a firewall rule to drop all pkts 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 pkts 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 [[firewall]](#firewall) pkt 3 receives no response and as such Python assumes that the pkt just got lost and as such tries to send the pkt again repeatedly, this continued for more than 30 seconds before a stopped it as shown by the time column in figure [[firewall]](#firewall) and the final KeyboardInterrupt in figure [[firewall\_code]](#firewall_code). 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.



Attempted connection to a closed port with and without firewall rule to drop pkts.



The code used to produce firewall pkt dropping example in figure [[firewall]](#firewall)

[firewall\_code]

Having explained firewalls, how they affect port scanning and other things above I will now explain what I am actually trying to achieve with my project and how I am going to do it. I am trying to make a tool similar to nmap which will be able to detect the state (as in whether the port is open/closed or filtered etc) of ports on remote machines, detect which hosts are up on a subnet and finally I want to be able to try to detect what services are listening behind any of the ports. I am going to be writing in Python version 3.7.2 as it is the latest stable release of Python 3 and has many features which are not in even fairly recent versions such as 3.5, the biggest one of these being fstrings which are where I can put a single a ‘f’ before a string and then any formatting options I put inside using curly braces are expanded and formatted accordingly. This allows for a clear and consistent string formatting syntax which I will use extensively. I will be using Python in particular as a language because it is very readable and has extensive low level bindings to C networking functions with the socket module allowing me to write code quickly which is easily understandable and has a clear purpose and at the same time be able to use low level networking functions and even changing the behaviour at this low level with socket.setsockopt. As well as this the socket module allows me to open sockets that communicate using many different protocols such as tcp, udp and icmp just to name a few. These features combine to make Python a great language for writing networking software with a high level of abstraction. In regards to the OSI model my code will sit with the user interface at level 7 specifying what to do at a high level then the actual scanning takes place at levels 3, 4 and 5 with host detection being at level 3. Port scanning will be taking place At level 4 for tcp SYN scanning and udp scanning. Whereas connect() scanning and version detection will sit at level 5. Finally I will look at what is actually handling all of the networking on my machine. My machine runs linux and as such all networking is handled by system calls to the linux kernel. For example the socket.connect method is just a call to the underlying linux kernel’s connect syscall but presenting a kinder call signature to the user as the Python socket library does some processing before the syscall is made.

## Success Criteria

1. Probe another computer’s networking from a bbox perspective.
2. Send icmp ECHO requests to determine whether a machine is active or not.
3. Translate cidr specified subnets into a list of domains.
4. Detect whether a TCP port is open (can be connected to).
5. Detect whether a TCP port is closed (will refuse connections).
6. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access).
7. Detect whether a UDP port is open (can be connected to).
8. Detect whether a UDP port is closed (will refuse connections).
9. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access).
10. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses.
11. Detect what service is listening behind a port.
12. Detect the version of the service running behind a port.

## Description of current system or existing solutions

Nmap is currently the most popular tool for doingportscanning and host enumeration. It supports the scanning types for determining information about remote hosts.

* tcp: SYN
* tcp: Connect()
* tcp: ACK
* tcp: Window
* tcp: Maimon
* tcp: Null
* tcp: FIN
* tcp: Xmas
* udp
* Zombie host/idle
* 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 pkts 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 ids such as sending pkts with spoofed csums/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/>). The following is an example nmap scan which I did on my home network: nmap -sC -sV -oA networkscan 192.168.1.0/24. Breaking it down this means to enable script scanning -sc, enable version detection -sV and then output all results in all the common formats: XML, nmap and greppable, using the base name networkscan which produces three files: networkscan.(nmap,gnmap,xml). Before I go into what each file contains I will explain some terminology, greppable is anything which can be easily searched with the linux grep which stands for Globally search a Regular Expression and Print, which basically means look in files for lines that contain a certain word or pattern, for example finding all lines with the word “hi” in them in the file “document” grep hi document. Onto the files: networkscan.nmap contains what would usually be printed by nmap while the scan is being run, it looks like this:

# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:  
 nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24  
Nmap scan report for router.asus.com (192.168.1.1)  
Host is up (1.0s latency).  
Not shown: 995 closed ports  
PORT STATE SERVICE VERSION  
53/tcp open domain (generic dns response: NOTIMP)  
| fingerprint-strings:   
| DNSVersionBindReqTCP:   
| version  
|\_ bind  
80/tcp open http ASUS WRT http admin  
|\_http-server-header: httpd/2.0  
|\_http-title: Site doesn't have a title (text/html).  
515/tcp open printer  
8443/tcp open ssl/http ASUS WRT http admin  
|\_http-server-header: httpd/2.0  
|\_http-title: Site doesn't have a title (text/html).  
| ssl-cert: Subject: commonName=192.168.1.1/countryName=US  
| Not valid before: 2018-05-05T05:05:17  
|\_Not valid after: 2028-05-05T05:05:17  
9100/tcp open jetdirect?  
1 service unrecognized despite returning data. If you know the service/version,  
please submit the following fingerprint at  
https://nmap.org/cgi-bin/submit.cgi?new-service :  
SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86\_64-pc-linux-gnu%r(DNSV  
SF:ersionBindReqTCP,20,"\0\x1e\0\x06\x85\x85\0\x01\0\0\0\0\0\0\x07version\  
SF:x04bind\0\0\x10\0\x03")%r(DNSStatusRequestTCP,E,"\0\x0c\0\0\x90\x04\0\0  
SF:\0\0\0\0\0\0");  
Service Info: CPE: cpe:/o:asus:wrt\_firmware

Above is just the report for one such device in the report as the full thing is over 200 lines lone. In it you can see information such as which ports are open and what services are running behind them as this is my router you can see port 8443 which nmap has recognised to be hosting the ASUS web admin from which you can configure the route. Then after than some other associated information extracted from the server. Most of this extra information is from the -sC flag which is script scanning and allows advanced interaction with running services specifically to gain more information by providing specialised probing per protocol. We can also see at the end an unrecognised service which nmap shows us the data it returned and asks us to submit a new service report at a given URL if we recognise the service. This system of submitting fingerprints of services is how nmap is so good at recognising services: it has a lot of data to look at and learn from in regards to service fingerprinting.

Next networkscan.gnmap:

# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:  
 nmap -sC -sV -oA /home/tritoke/networkscan 192.168.1.0/24  
Host: 192.168.1.1 (router.asus.com) Status: Up  
Host: 192.168.1.1 (router.asus.com) Ports: 53/open/tcp//domain//  
 (generic dns response: NOTIMP)/, 80/open/tcp//http//ASUS WRT http admin/,  
 515/open/tcp//printer///, 8443/open/tcp//ssl|http//ASUS WRT http admin/,  
 9100/open/tcp//jetdirect?/// Ignored State: closed (995)  
Host: 192.168.1.8 (android-25a97e36c2e74456) Status: Up  
Host: 192.168.1.8 (android-25a97e36c2e74456) Ports: 5060/filtered/tcp//sip///  
 Ignored State: closed (999)

Again this is not all of the file as it is very large. As you can see above all of the information is on a single line for each type of scan, this is useful if you want to scan a large number of hosts and just want to know which hosts are up you can do grep 'Status: Up' networkscan.gnmap which outputs this:

$ grep 'Status: Up' networkscan.gnmap  
Host: 192.168.1.1 (router.asus.com) Status: Up  
Host: 192.168.1.8 (android-25a97e36c2e74456) Status: Up  
Host: 192.168.1.10 (diskstation) Status: Up  
Host: 192.168.1.88 () Status: Up  
Host: 192.168.1.88 () Status: Up  
Host: 192.168.1.117 () Status: Up  
Host: 192.168.1.159 (groot) Status: Up  
Host: 192.168.1.159 (groot) Status: Up  
Host: 192.168.1.176 (ET0021B7C01F2E) Status: Up

Showing you clearly the hosts which are online and then their host names. Other ways to use this output format would be to find out which ports are open on only one machine, or which hosts have a webserver running on them or a vulnerable version of a mail server etc. In general it is useful for when you want to filter results.

Finally we have xml format:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE nmaprun>  
<?xml-stylesheet href="file:///usr/bin/../share/nmap/nmap.xsl" type="text/xsl"?>  
<!-- Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as: nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24 -->  
<nmaprun scanner="nmap" args="nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24" start="1554921378" startstr="Wed Apr 10 19:36:18 2019" version="7.70" xmloutputversion="1.04">  
<verbose level="0"/>  
<debugging level="0"/>  
<host starttime="1554921379" endtime="1554923187"><status state="up" reason="syn-ack" reason\_ttl="0"/>  
<address addr="192.168.1.1" addrtype="ipv4"/>  
<hostnames>  
<hostname name="router.asus.com" type="PTR"/>  
</hostnames>  
<ports><extraports state="closed" count="995">  
<extrareasons reason="conn-refused" count="995"/>  
</extraports>  
<port protocol="tcp" portid="53"><state state="open" reason="syn-ack" reason\_ttl="0"/><service name="domain" extrainfo="generic dns response: NOTIMP" servicefp="SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86\_64  
-pc-linux-gnu%r(DNSVersionBindReqTCP,20,&quot;\0\x1e\0\x06\x85\x85\0  
\x01\0\0\0\0\0\0\x07version\x04bind\0\0\x10\0\x03&quot;)%r  
(DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\0\0\0\0\0\0\0\0&quot;);" method="probed" conf="10"/><script id="fingerprint-strings" output="&#xa; DNSVersionBindReqTCP: &#xa; version&#xa; bind"><elem key="DNSVersionBindReqTCP">&#xa; version&#xa; bind</elem>  
</script></port>

It is verbose in the extreme contains the reason why each port has the state it does as well as a vast amount of other data that the other scans didn’t include as well as this it is not very human readable meaning that this format is more likely available because it is easier for other programs to parse than the other formats. As well as this the verbosity can be good if you really need to dive into why a port was marked as closed etc or the exact bytes that a service replied with.

In terms of where nmap lives in the software stack is that it is an application at level 7 when the user interacts with it but it uses several libraries which interact at level 2 which it uses to get the raw headers of the packets being sent and thus gain information from them.

## 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)

## Data Dictionary

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

## Data Flow Diagram

[https://en.wikipedia.org/wiki/Data-flow\_diagram](https://en.wikipedia.org/wiki/Data-flow_diagram<Paste>)

## Data Sources

## Description of Solution Details, OOP/Mobile/Networking

To do all forms of scanning other than Connect() scanning and version detection, custom pkts are made to allow the half open (no full connection is made to the host) scanning used intcpSYN scanning. Making custom pkts 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 pkt is interpreted by the network switch, for example theipaddr192.168.1.58 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 pkt is not routed to the correct host. As well as the issues with byte order and the interpretation of information at different points the csum which is embedded into the pkt 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 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.

## 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 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

## Data Volumes (Supplementary)

<https://stackoverflow.com/questions/5566841/what-are-data-volumes#5567390>

## 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 pkts going over the wire, as well as this it has a vast array of pkt decoders (2231 in my install) which it can use to dissect almost any pkt that would be on the network. The main benefit of wireshark is that I can see my scanners sending pkts and then check whether the parsers that I have written for the different protocols are working. I can also check that the csums in each of the various protocols is valid as wireshark does csum verification for various protocols.

# Design

## 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 pkts to. It can then differentiate between open, open|filtered, filtered and closed ports based on whether it receives a pkt back and what flags (part oftcp pkts are a one byte long section which store “flags” where each bit in the byte represents a different flag) are set in the received pkt.

## 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.

## System Algorithms (Flowcharts)

## 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 anipaddris converted between “long form” and “dot form” which is used in every type of scanning.

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

[ip\_range]

## Design Data Dictionary

# Technical Solution

## Program Listing

## Comments (Core)

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

# Testing

## Test Plan

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

# Evaluation

## Reflection on final outcome

## Evaluation against objectives, end user feedback

## Potential improvements

# Appendices

You may show you program listing here

User feedback and survey data