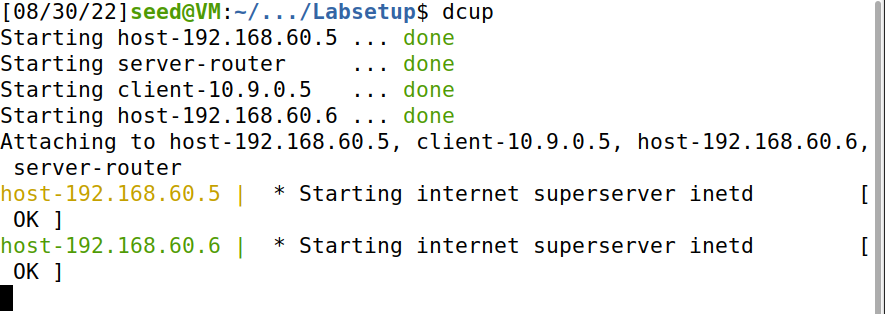
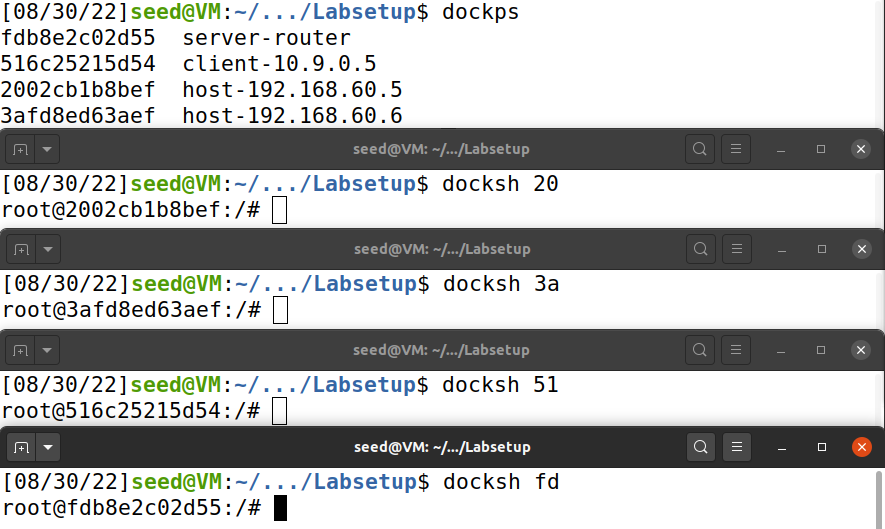
**Fianl Task – More Complete VPN**

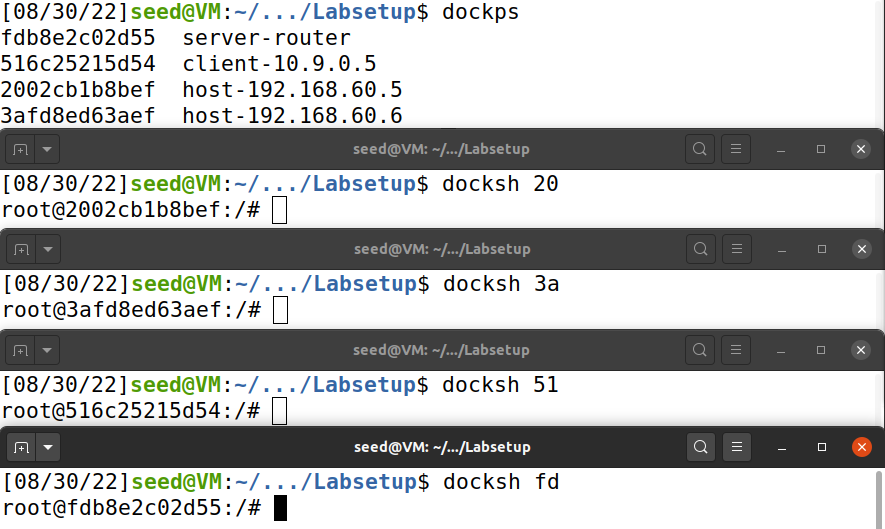
**环境配置**

仍然使用Lab4中所使用的容器



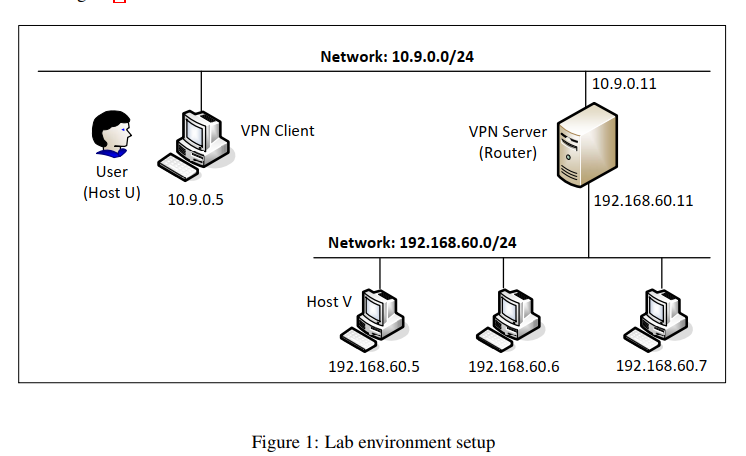
另外开启多个终端，输入docksh xx来连接到对应容器：





只需要连接容器client（10.9.0.5）、server（10.9.0.11/192.168.60.11）和host（192.168.60.5）

实验网络拓扑如下：



1. **VPN原理介绍**

虚拟专用网络（VPN）是建立在公共网络（通常是互联网）之上的专用网络。

VPN内的计算机可以安全通信，就像它们在物理上与外部隔离的真正的专用网络上一样，即使它们的流量可能通过公共网络。VPN使员工能够在旅行时安全访问公司的内部网；它还允许公司将其私人网络扩展到全国各地和世界各地。

现实中，VPN客户端和VPN服务器通过互联网连接。为了简单起见，我们在实验中直接将这两台机器连接到同一个局域网，即该局域网模拟互联网。

第三台机器，主机V（192.168.60.5），是专用网络中的一台计算机。主机U（10.9.0.5，专用网络之外）上的用户希望通过VPN隧道与主机V通信。为了模拟这种设置，我们将主机V连接到VPN服务器（也用作网关）。在这种设置中，主机V不能直接从互联网访问；也不能从主机U直接访问。

1. **程序设计方案**

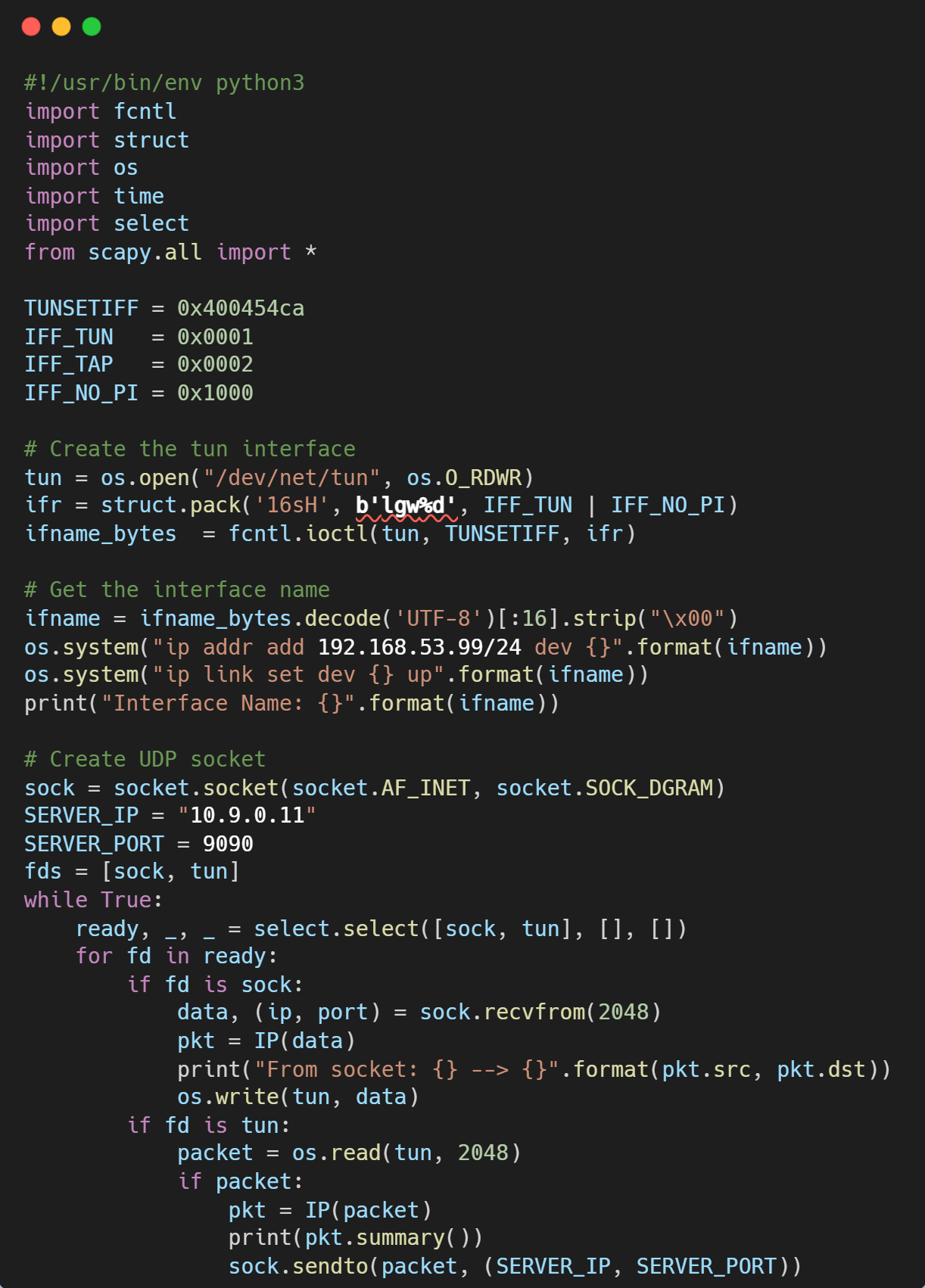
本次实验主要基于Lab4中已经实现的简易VPN程序。在其基础上增加一些功能，更便于使用，并且使其具有一些交互性。

所设计的方案如下：

1. 以原python程序作为运行中的核心模块，来实现VPN的主要功能；
2. 添加配置文件，以便于修改，其中所包括的项应该有：网卡名称、网卡地址、VPN服务器地址、VPN服务器端口号、目的网络的地址；
3. 相应地，程序需要有对配置文件的读、写和加载功能，并使用从配置文件中加载的值来替换原程序中的内容；
4. 解决需要用户手动在shell中添加路由表项的问题，改为在程序中自动配置；
5. 设计一个简易界面用来与用户进行交互；
6. ~~在客户端对用户发送/接收的包中的数据进行加密/解密，在服务器端对发至专用网络的包中的数据进行解密、对发送至VPN客户端的的包中的数据进行加密，以保证在互联网中传输的安全性。~~**注：这一部分暂时没有实现。**
7. **各模块的实现**
8. **VPN核心运行模块的实现**

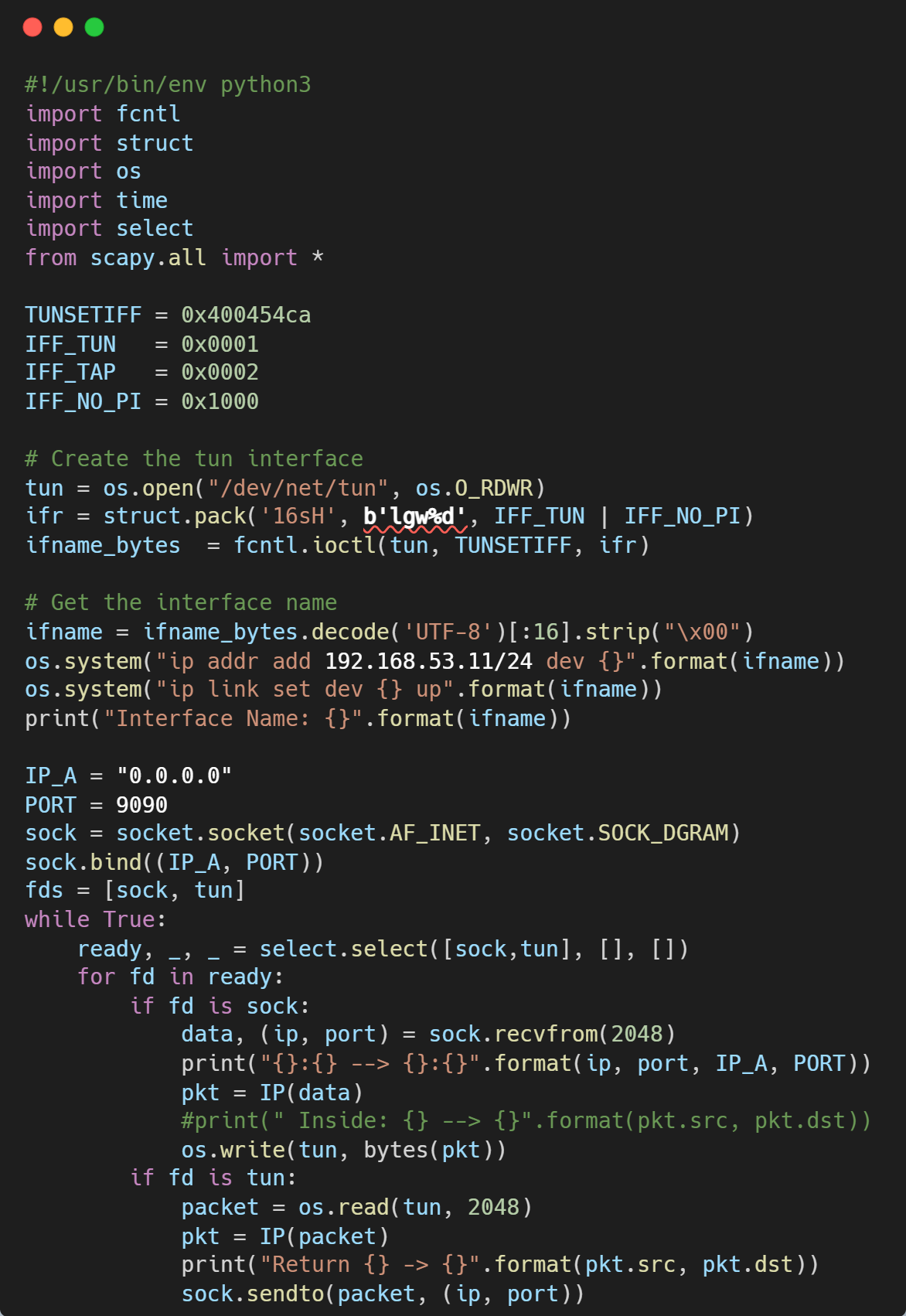
分为客户端程序和服务器端程序。

**客户端：**



其中白色字体部分是可以在接下来要实现的配置文件中加载替换的部分，红色波浪线字体是需要与服务器端程序中保持一致的部分（网卡名称）。

**服务器端：**

****

其中白色字体部分也是可以用于从配置文件中读取替换的，红色波浪线字体是需要和客户端程序保持一致的。

一开始的设计是在服务器端也使用配置文件的形式，但是考虑到现实中用户没有办法去操作服务器来修改这些信息，但是只在用户端修改信息而服务器端不同步会导致错误，所以设想是客户端首先与服务器建立通信，发送服务器所需要的信息，服务器收到后使用这些信息来更改自己的配置文件。但是这个过程过于复杂，而且还应当考虑到双方的相互认证等，难以实现，最后在服务器端还是使用手动配置的方式。

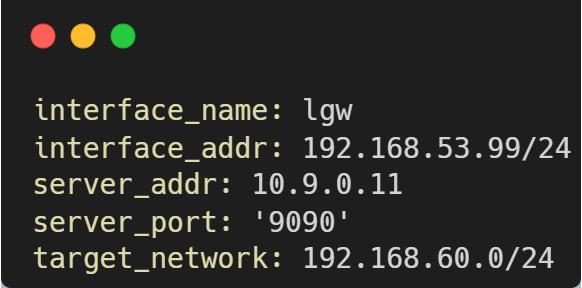
1. **配置文件读写与加载的实现**

**配置文件创建：**

配置文件这里使用yaml格式，首先需要在容器中shell输入*pip install pyyaml*来安装yaml，否则在python中import会报错。

按照前面的设计，配置文件中包含：网卡名称、网卡地址、VPN服务器地址、VPN服务器端口号、目的网络的地址。

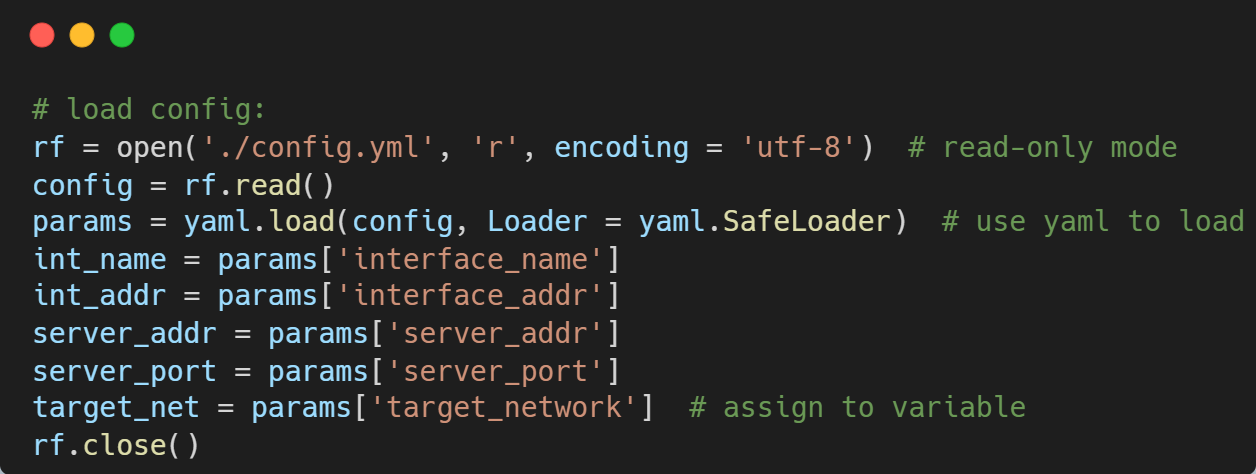
在程序所在目录下创建config.yml，内容、格式如下：



（图中是在程序中写入过的，类型不一，按理说应当只有端口号类型为int，其他都应为string）

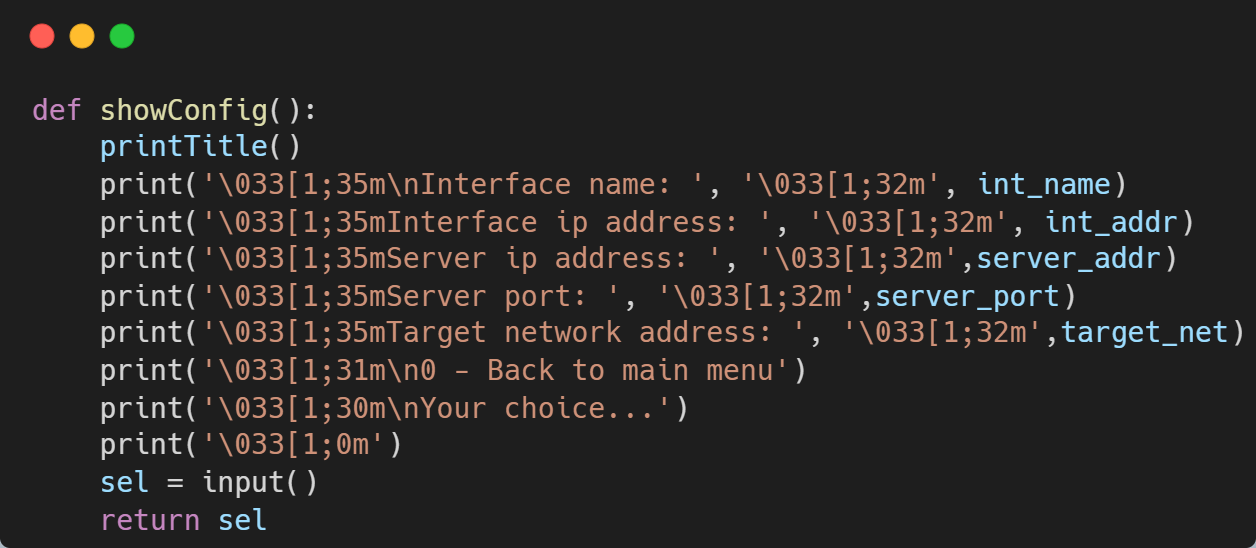
**配置文件读取/加载：**

以只读模式和utf-8编码方式打开配置文件，使用yaml从中加载，并赋值给相应的变量，之后关闭文件，这些变量都作为全局变量，便于其他函数的调用：

****

**打印配置信息：**

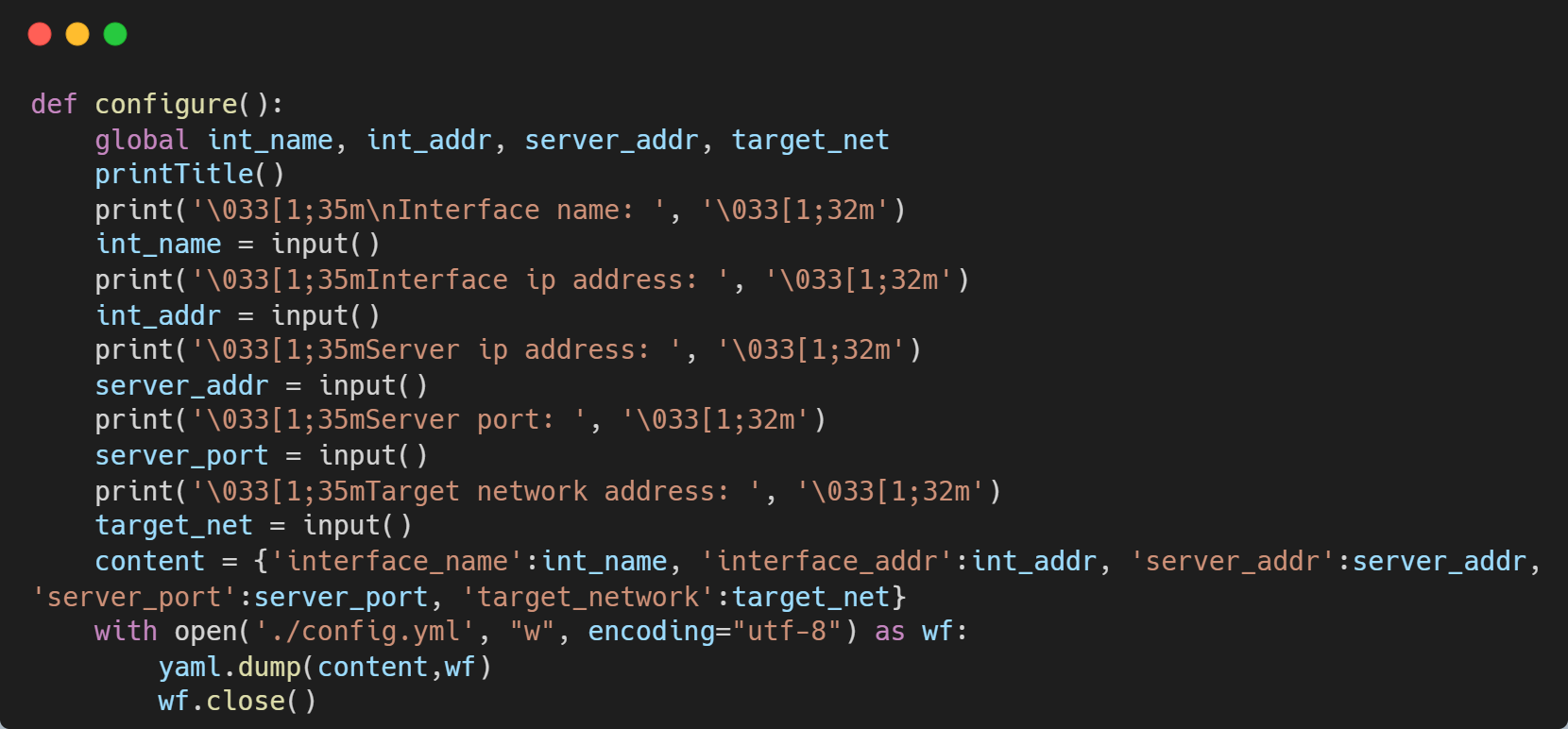
将配置文件中的信息打印在shell中，并且添加了一些颜色效果：

****

printTitle()是后面用来打印程序标题的一个函数，print()中的\033等是对输出的格式进行调整，包括字体、颜色等，sel是用于在后面整个程序运行中判断用户的输入进行循环或退出的变量。

**写入/更改配置信息：**

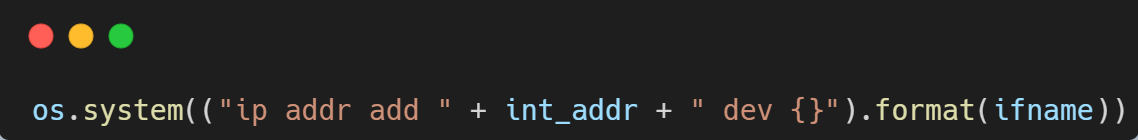
在程序提示下，用户输入新的配置信息，程序以覆盖模式写入配置文件，关闭文件：

****

在测试中发现，可以继续完善功能，让用户选择所要更改的条目，而不是需要用户从头到尾全部重新配置一遍。

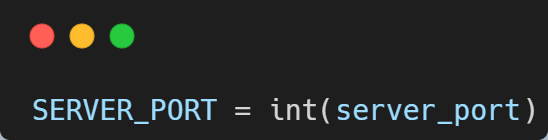
替换客户端程序中的相应字段：

几个ip地址可以直接使用加号‘+’连接进字符串中，比如：

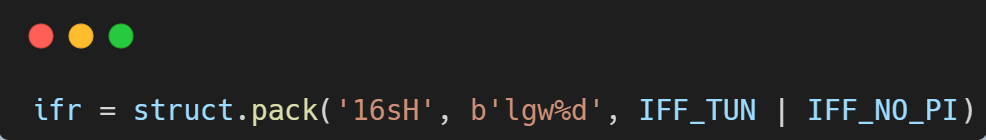


而以下两个需要注意：

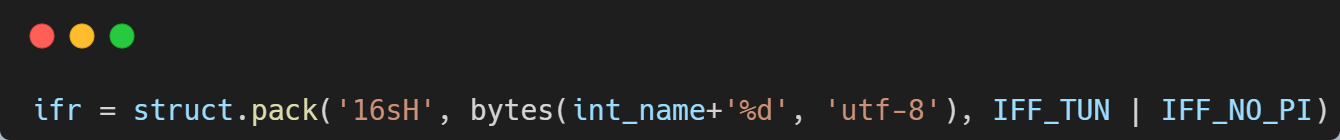
VPN服务器端口号需要转为int类型再赋值：



网卡名称在原程序中可以看到是写为以下形式的：



查阅资料后，代码中b’’的作用是将string转为bytes类型，与之功能相同的函数是bytes()，所以网卡名称部分修改为如下：

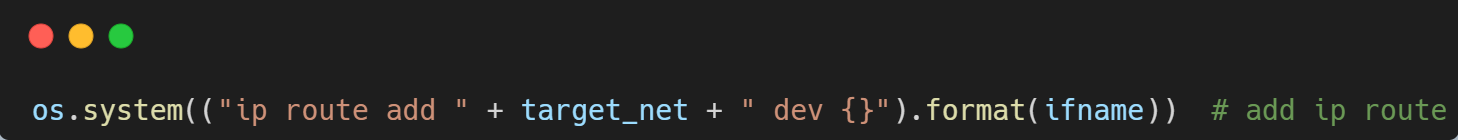


1. **程序自动添加路由表项的实现**

在Lab4实验过程中，需要在client容器中多次手动输入*ip route add 192.168.60.0/24 dev lgw0（自定义网卡名）*，不仅麻烦而且容易忘记这个步骤或者输入错误，因此想到将这个命令添加进程序中自动执行。

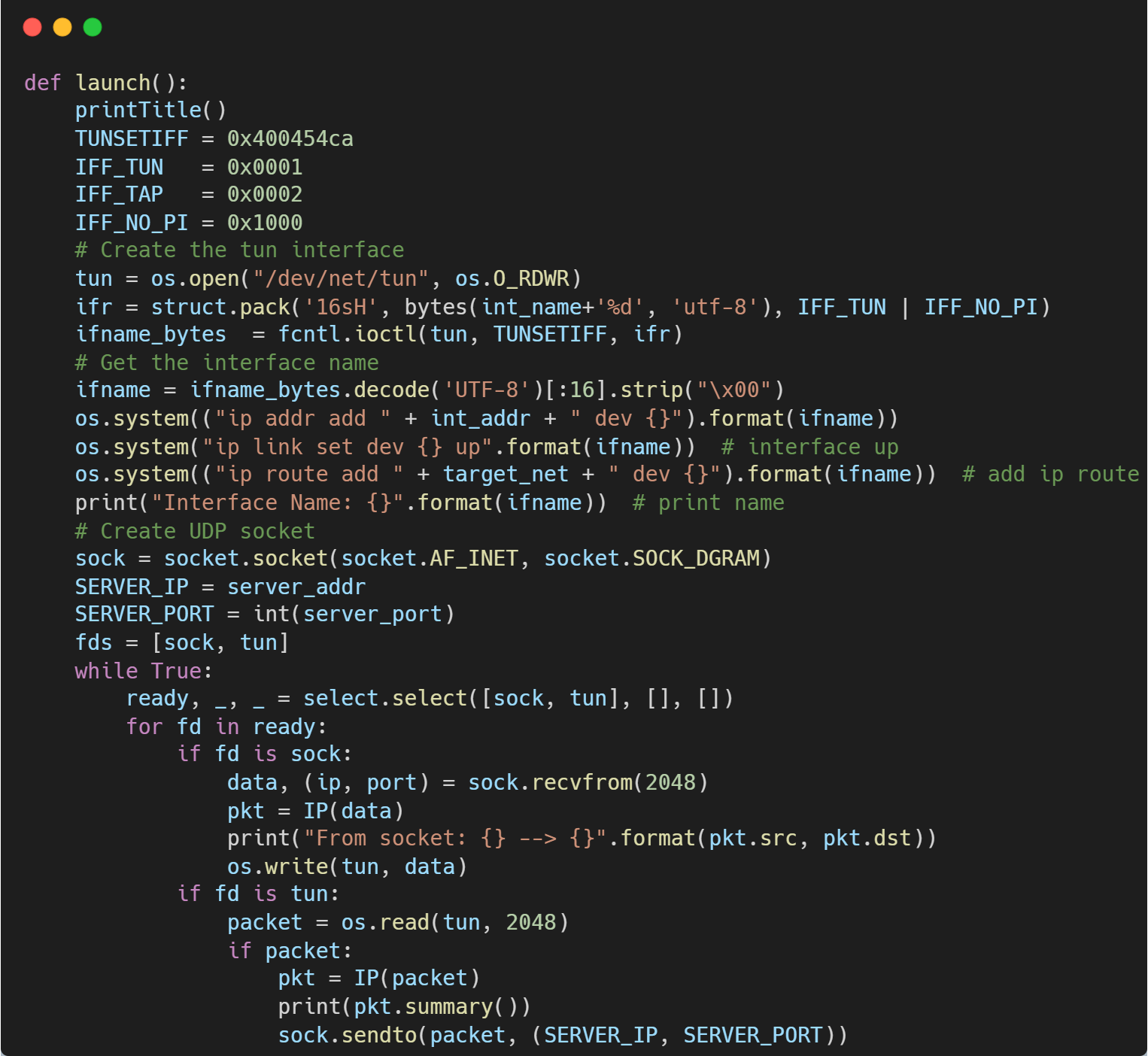
查阅相关资料后，可以使用os.system()函数来实现执行命令，需要import os。

将命令中的*192.168.60.0/24*使用从配置文件中加载的target\_net（目标网络的地址）来替换，网卡名称用程序中已经转换好的值替换：



在测试程序时，发现会报出设备未开启的错误。发现错误原因是上面一句命令添加的位置不对，正确的顺序应当是在开启网卡的命令执行后再添加路由表项。

此时的VPN客户端的主要实现函数如下：



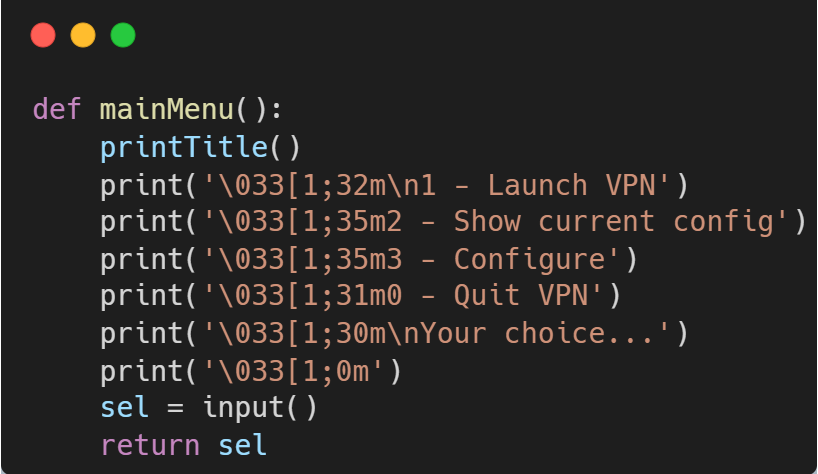
1. **简易界面的实现**

首先对shell进行清屏操作，在linux中，使用*clear*命令清屏，使用前面的os.system()方法来执行这个命令。

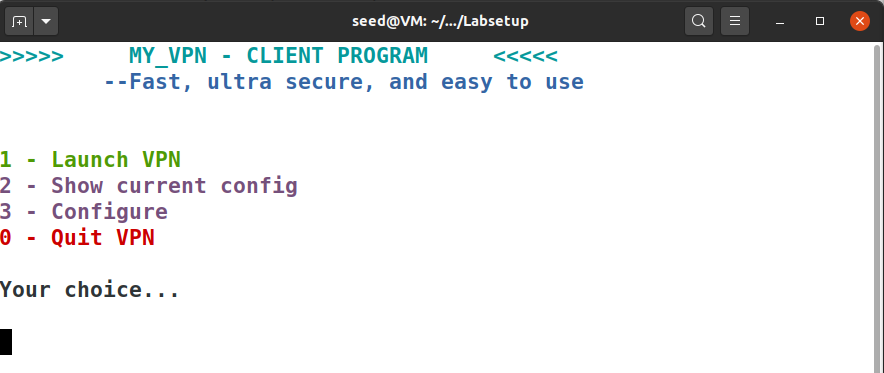
设计一个标头，让它看起来像一个有模有样的程序：



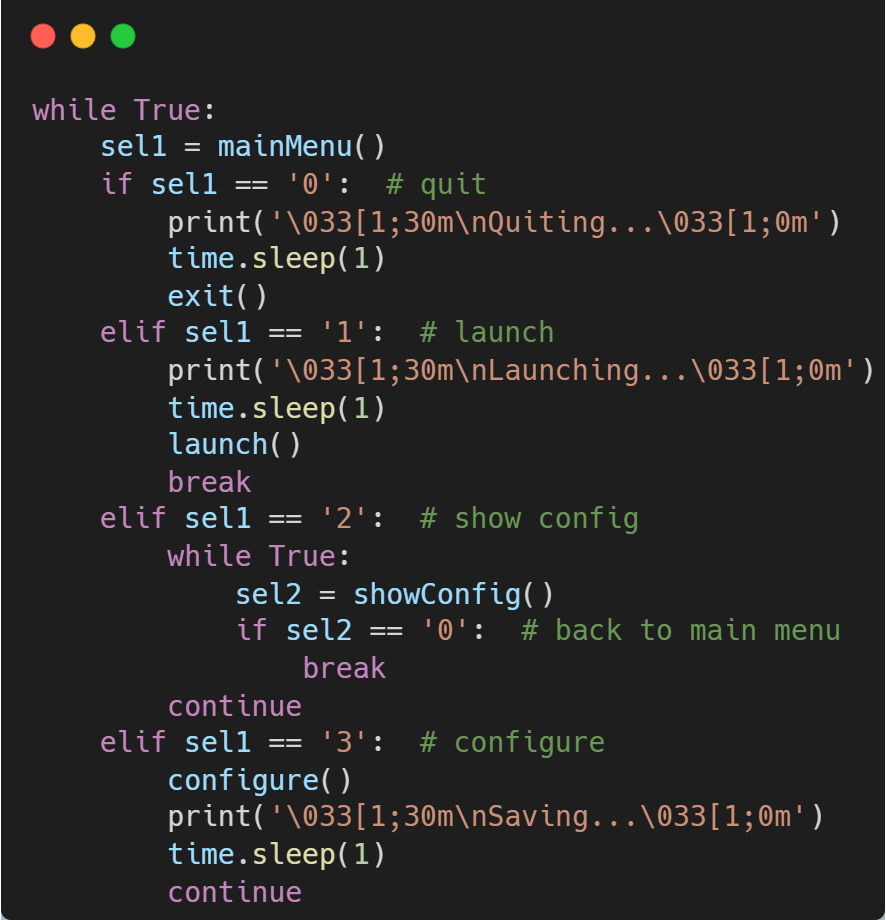
然后是主菜单的制作，在标头下面添加一些选项，比如：启动程序、退出程序、修改配置文件、打印配置信息等：



此时程序的界面是这样的：



下面添加好对用户输入的判断，对于不同的输入跳转到不同的函数来执行，以及循环的继续或退出：



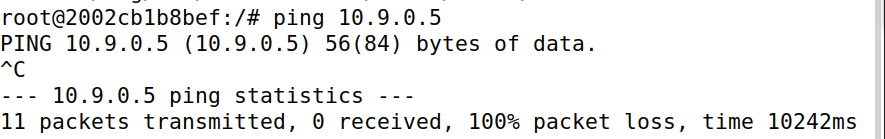
1. **程序整体代码展示**
2. #!/usr/bin/env python3
4. **import** os
5. **import** time
6. **import** yaml
7. **import** fcntl
8. **import** struct
9. **import** time
10. **import** select
11. **from** scapy.all **import** \*

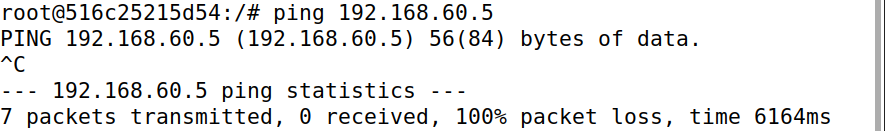
14. **def** printTitle():
15. os.system('clear')
16. **print**('\033[1;36m>>>>>     MY\_VPN - CLIENT PROGRAM     <<<<<')
17. **print**('\033[1;34m\t--Fast, ultra secure, and easy to use')
18. **print**('\033[1;0m')
20. **def** mainMenu():
21. printTitle()
22. **print**('\033[1;32m\n1 - Launch VPN')
23. **print**('\033[1;35m2 - Show current config')
24. **print**('\033[1;35m3 - Configure')
25. **print**('\033[1;31m0 - Quit VPN')
26. **print**('\033[1;30m\nYour choice...')
27. **print**('\033[1;0m')
28. sel = input()
29. **return** sel
31. **def** launch():
32. printTitle()
33. TUNSETIFF = 0x400454ca
34. IFF\_TUN   = 0x0001
35. IFF\_TAP   = 0x0002
36. IFF\_NO\_PI = 0x1000
37. # Create the tun interface
38. tun = os.open("/dev/net/tun", os.O\_RDWR)
39. ifr = struct.pack('16sH', bytes(int\_name+'%d', 'utf-8'), IFF\_TUN | IFF\_NO\_PI)
40. ifname\_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
41. # Get the interface name
42. ifname = ifname\_bytes.decode('UTF-8')[:16].strip("\x00")
43. os.system(("ip addr add " + int\_addr + " dev {}").format(ifname))
44. os.system("ip link set dev {} up".format(ifname))  # interface up
45. os.system(("ip route add " + target\_net + " dev {}").format(ifname))  # add ip route
46. **print**("Interface Name: {}".format(ifname))  # print name
47. # Create UDP socket
48. sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)
49. SERVER\_IP = server\_addr
50. SERVER\_PORT = int(server\_port)
51. fds = [sock, tun]
52. **while** True:
53. ready, \_, \_ = select.select([sock, tun], [], [])
54. **for** fd **in** ready:
55. **if** fd **is** sock:
56. data, (ip, port) = sock.recvfrom(2048)
57. pkt = IP(data)
58. **print**("From socket: {} --> {}".format(pkt.src, pkt.dst))
59. os.write(tun, data)
60. **if** fd **is** tun:
61. packet = os.read(tun, 2048)
62. **if** packet:
63. pkt = IP(packet)
64. **print**(pkt.summary())
65. sock.sendto(packet, (SERVER\_IP, SERVER\_PORT))
67. **def** showConfig():
68. printTitle()
69. **print**('\033[1;35m\nInterface name: ', '\033[1;32m', int\_name)
70. **print**('\033[1;35mInterface ip address: ', '\033[1;32m', int\_addr)
71. **print**('\033[1;35mServer ip address: ', '\033[1;32m',server\_addr)
72. **print**('\033[1;35mServer port: ', '\033[1;32m',server\_port)
73. **print**('\033[1;35mTarget network address: ', '\033[1;32m',target\_net)
74. **print**('\033[1;31m\n0 - Back to main menu')
75. **print**('\033[1;30m\nYour choice...')
76. **print**('\033[1;0m')
77. sel = input()
78. **return** sel
80. **def** configure():
81. **global** int\_name, int\_addr, server\_addr, target\_net
82. printTitle()
83. **print**('\033[1;35m\nInterface name: ', '\033[1;32m')
84. int\_name = input()
85. **print**('\033[1;35mInterface ip address: ', '\033[1;32m')
86. int\_addr = input()
87. **print**('\033[1;35mServer ip address: ', '\033[1;32m')
88. server\_addr = input()
89. **print**('\033[1;35mServer port: ', '\033[1;32m')
90. server\_port = input()
91. **print**('\033[1;35mTarget network address: ', '\033[1;32m')
92. target\_net = input()
93. content = {'interface\_name':int\_name, 'interface\_addr':int\_addr, 'server\_addr':server\_addr, 'server\_port':server\_port, 'target\_network':target\_net}
94. with open('./config.yml', "w", encoding="utf-8") as wf:
95. yaml.dump(content,wf)
96. wf.close()

99. # load config:
100. rf = open('./config.yml', 'r', encoding = 'utf-8')
101. config = rf.read()
102. params = yaml.load(config, Loader = yaml.SafeLoader)
103. int\_name = params['interface\_name']
104. int\_addr = params['interface\_addr']
105. server\_addr = params['server\_addr']
106. server\_port = params['server\_port']
107. target\_net = params['target\_network']
108. rf.close()
110. **while** True:
111. sel1 = mainMenu()
112. **if** sel1 == '0':  # quit
113. **print**('\033[1;30m\nQuiting...\033[1;0m')
114. time.sleep(1)
115. exit()
116. **elif** sel1 == '1':  # launch
117. **print**('\033[1;30m\nLaunching...\033[1;0m')
118. time.sleep(1)
119. launch()
120. **break**
121. **elif** sel1 == '2':  # show config
122. **while** True:
123. sel2 = showConfig()
124. **if** sel2 == '0':  # back to main menu
125. **break**
126. **continue**
127. **elif** sel1 == '3':  # configure
128. configure()
129. **print**('\033[1;30m\nSaving...\033[1;0m')
130. time.sleep(1)
131. **continue**
132. **功能测试**

**在程序运行前，首先测试网络状况：**

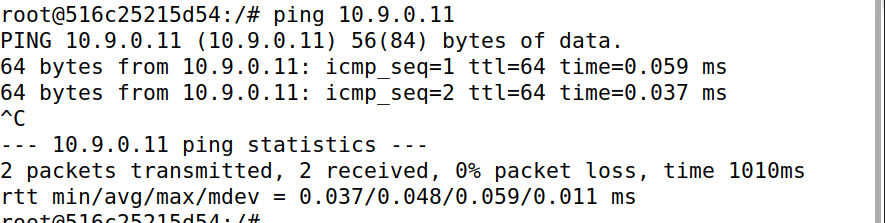
测试client（10.9.0.5）与host（192.168.60.5）之间的连通性：

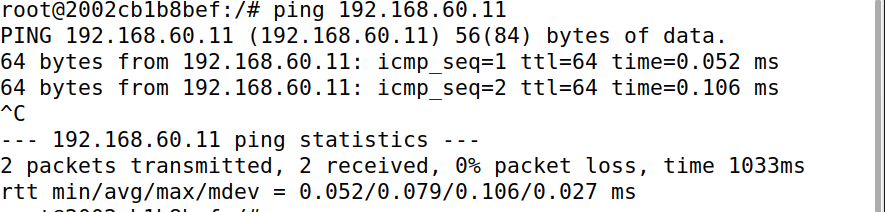




两边均是不通的。

测试client（10.9.0.5）与host（192.168.60.5）分别到各自网关的连通性：



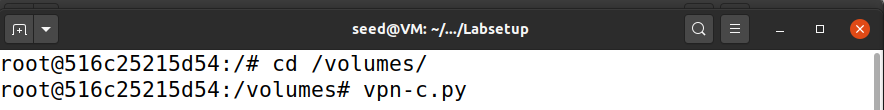


都能够成功ping通。

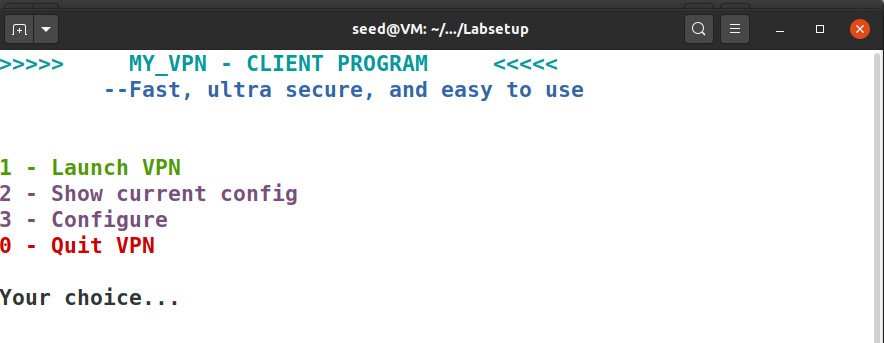
网络状况测试完成。

**测试程序功能：**

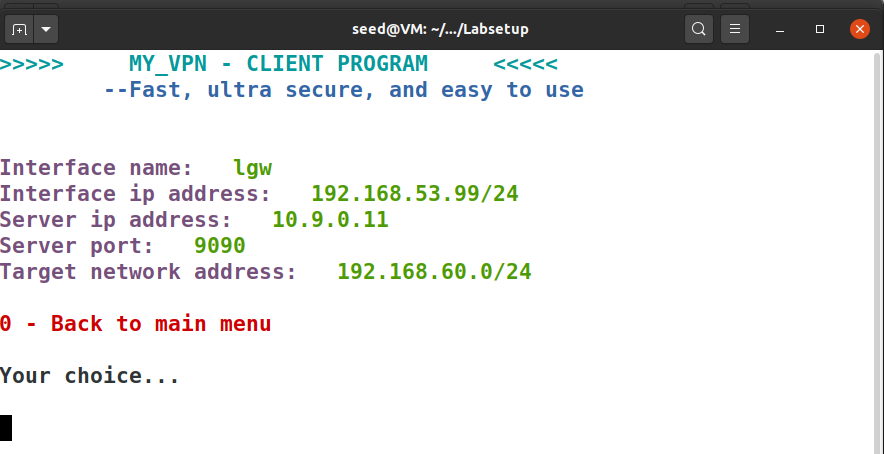
Client容器进入/volumes路径下运行vpn-c.py程序：



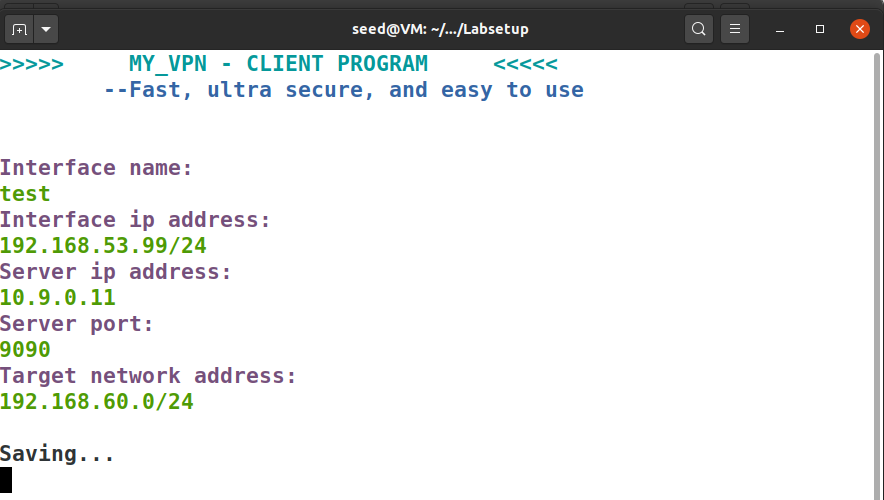
程序界面如下：



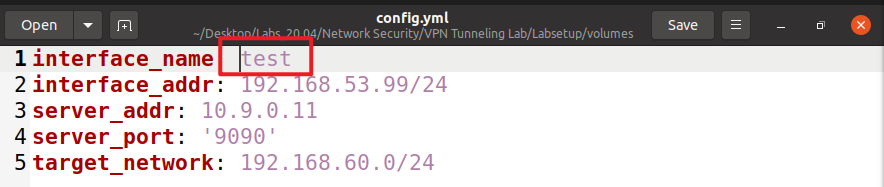
输入2，查看当前配置信息：



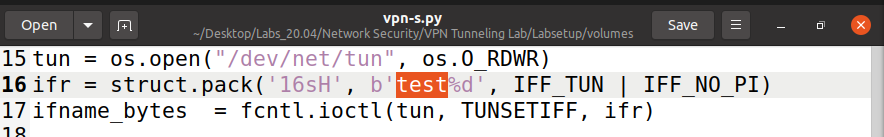
输入0，返回主菜单。输入3，修改配置文件，这里只修改一下网卡名称为test，其他均不修改：



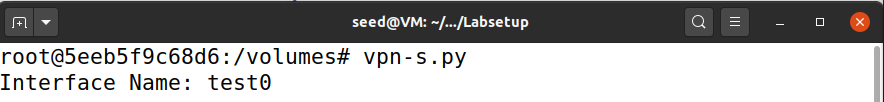
在文本编辑器中查看config.yml文件，已被修改：



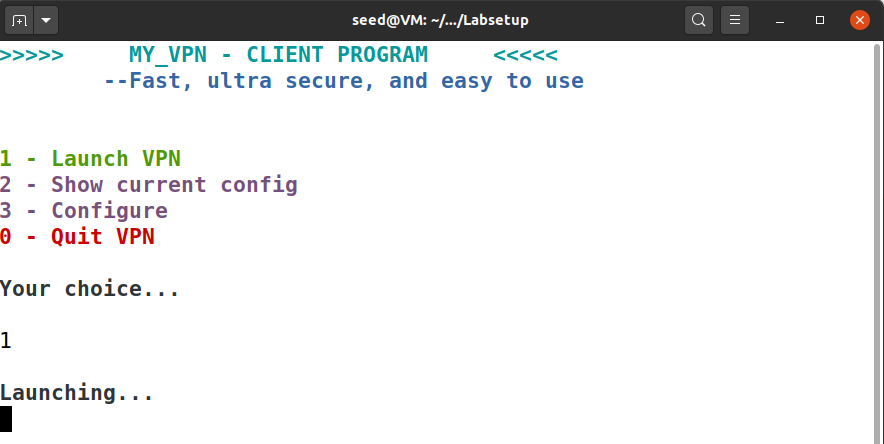
这里需要手动修改一下vpn-s.py的网卡名称为test：

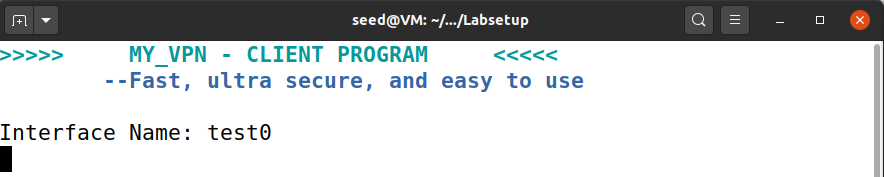


在server（10.9.0.11/192.168.60.11）上运行vpn-s.py：

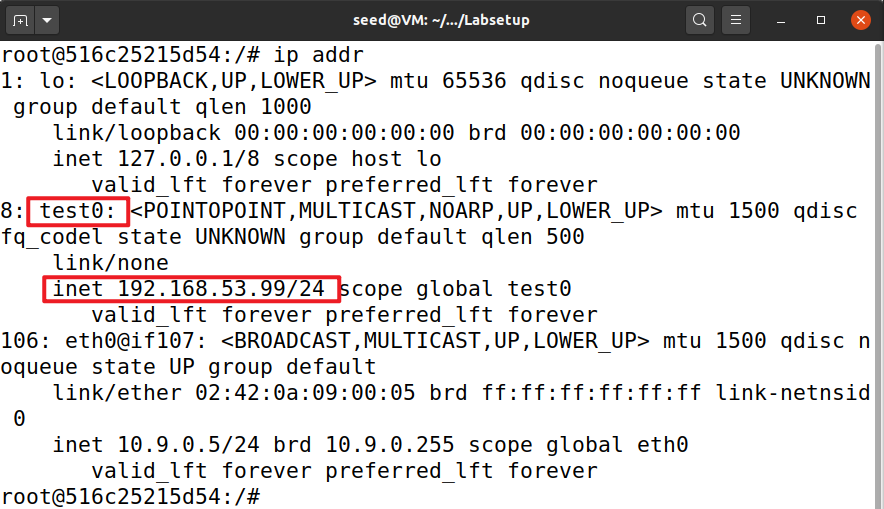


在vpn-c程中输入1，运行起VPN：



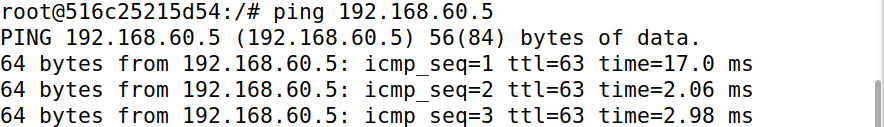


另开一个shell连接到client容器，输入*ip addr*查看网络信息：

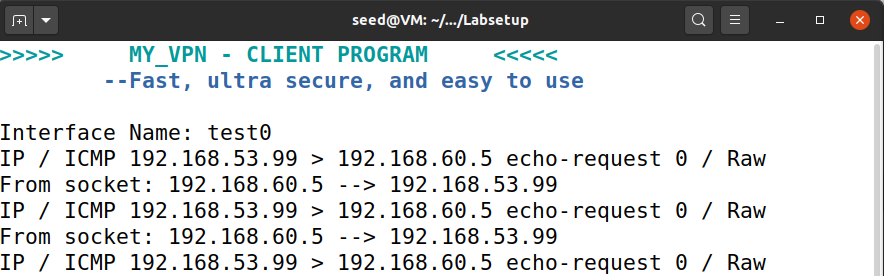


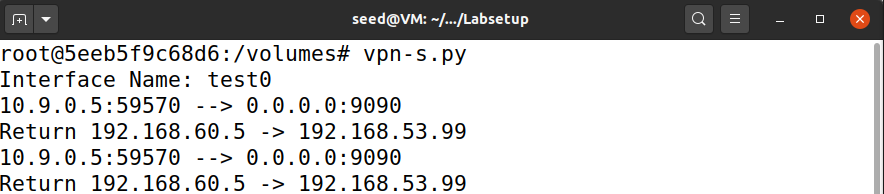
可以看到网卡信息与我们所配置的相同。

输入ping 192.168.60.5，与host建立连接：

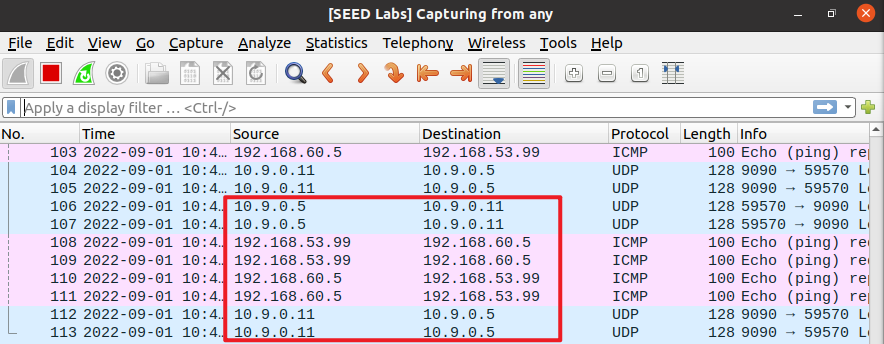


成功ping通，且在vpn-s与vpn-c程序中也有相应输出：



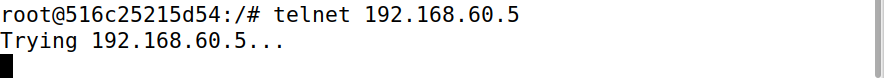


在wireshark中查看捕获的流量：

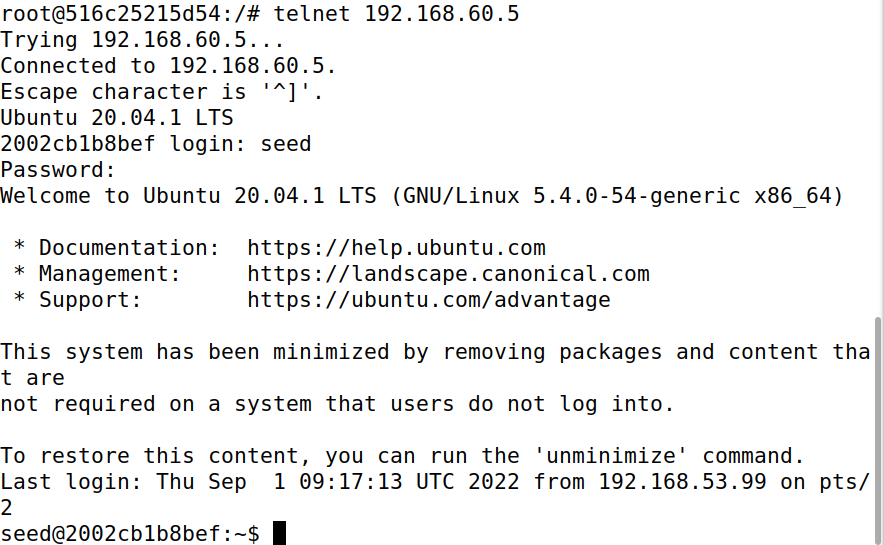


红框中即为从client到host再返回client的一个来回。经过VPN Server的转发。

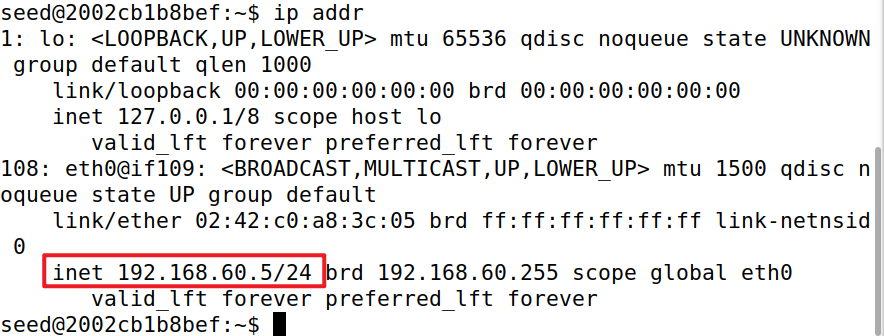
结束vpn-s的运行后，尝试在client上访问host的telnet服务：



是不成功的，再次运行起vpn-s后，等待一小段时间后，出现了登录的输出，输入用户名seed与口令dees后成功登录：

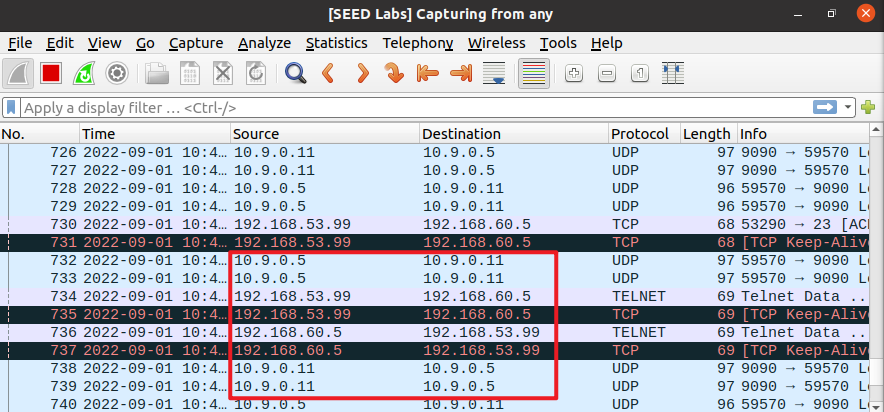


根据id：2002cb1b8bef可以看出确实是host主机，输入ip addr命令，看到地址是192.168.60.5/24，证明成功访问了host的telnet服务：



输入ctrl + ]后输入quit退出telnet。

Wireshark中抓取的内容与前面相似，是相同的路径：



1. **不足、改进措施与收获**

本次的程序设计还有很多能够进一步完善的地方：

比如对于client与host之间在公共网络上发送的数据需要进行加密与解密来保证通信的安全；

还有前面提到的服务端的配置信息可以通过client发送给server，而server只需要保持一个监听程序的运行，在收到请求后，根据请求内容配置并运行vpn-s程序；

以及客户端程序的一些交互功能可以做的更加人性化，比如前面提到的配置的修改部分。

以上一些未实现的功能在最初设想时是很完美的，但需要去在代码中实现起来的时候就会发现各种各样的困难，有太多的内容需要去学习而不是简单的照抄照搬，在能力、时间都相当有限的情况下没能一一实现。

同时也通过暑期学校的这一系列课程与实验学习到了很多之前听过或使用过，但没用深入了解过的工具、协议等的原理，包括一些经常能够听到的攻击类型在代码层面是如何实现的、还有像本实验中的VPN，经常使用但没有了解过原理。在这些课程中，能够有机会去深入了解并自己实现，相信会对以后的学习工作有很大帮助。