

---

# 华中科技大学

## “计算机网络安全”实验报告

### 题目： VPN 实验

院 系 网络空间安全学院

专业班级 信安 1805 班

姓 名 吴 锦 添

学 号 U201810398

日 期 2022 年 5 月 30

评分项	实验报告评分 (40%)	检查单分数 (60%)	综合得分	教师签名
得分				

## 实验报告评分标准

评分项目	分值	评分标准	得分
实验原理	20	18-20 系统流程清晰，报文处理过程描述清楚； 15-17 系统流程比较清晰，报文处理过程描述比较清楚； 12 以下 描述简单	
实验步骤	30	25-30 实验步骤描述详细、清楚、完整，前后关系清晰； 18-24 实验步骤描述比较清楚，关键步骤都进行了描述 18 分以下，实验步骤描述比较简单或不完整	
结果验证与分析	20	16-20，任务完成，针对任务点的测试，对结果有分析 10-15，针对任务点的测试截图，没分析 10 分以下，测试很简单，没有覆盖任务点	
心得体会	10	8-10 有自己的真实体会 4-7 真实体会套话 3 分以下，没有写什么体会	
格式规范	10	图、表的说明，行间距、缩进、目录等，一种不规范扣 1 分	
实验思考	10	思考题的回答，以及其它的简介	
总 分			

---

## 目 录

<b>实验三 VPN 实验</b> .....	1
1 实验目的 .....	1
2 实验环境 .....	1
3 实验内容.....	2
4 实验步骤及结果分析 .....	3
5 实验思考 .....	19
<b>心得体会与建议</b> .....	21
1 心得体会 .....	21
2 建议 .....	22

---

## 实验三 VPN 实验

### 1 实验目的

本实验的学习目标是掌握 VPN 的网络和安全技术。为实现这一目标，要求实现简单的 TLS/SSL VPN。虽然这个 VPN 很简单，但它包含了 VPN 的所有基本元素。TLS/SSL VPN 的设计和实现体现了许多安全原则，包括以下内容：

- 虚拟专用网络
- TUN/TAP 和 IP 隧道
- 路由
- 公钥加密，PKI 和 X.509 证书
- TLS/SSL 编程
- 身份认证

### 2 实验环境

**实验机器的操作系统：**

VMware Workstation 虚拟机。

Ubuntu 16.04 操作系统（SEEDUbuntu16.04）。

**实验的网络配置：**

使用 VMware Workstation 虚拟机的 NAT 模式。

实验中所需的内网、外网，在相应的 docker 中进行网络配置即可。

**系统软件组件：**

本次实验需要使用 openssl 软件，该软件包含头文件，库函数和命令。该软件包已经安装在我们上述 VM 镜像中。

网络拓扑:

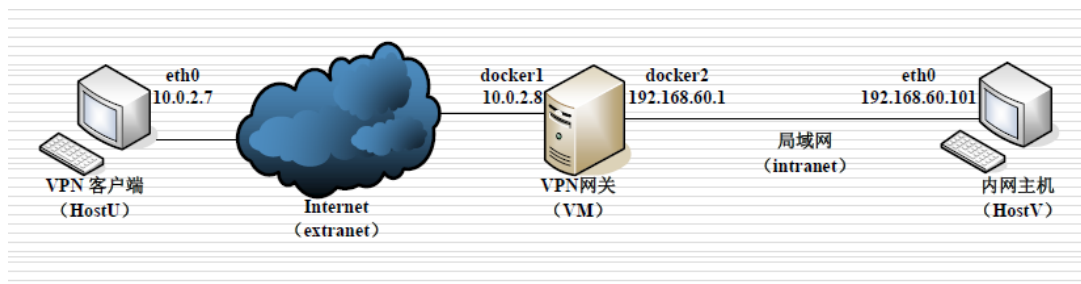


图 2.1 网络拓扑图

### 3 实验内容

#### 任务一：建立主机到主机的隧道

1. 在计算机（客户端）和网关之间创建 VPN 隧道，允许计算机通过网关安全地访问专用网络。我们使用虚拟机本身作为 VPN 服务器网关（VM），并创建两个容器分别作为 VPN 客户端（HostU）和专用网络中的主机（HostV）。
2. 同时使用指令配置内网和外网网段和相应的网关 IP，通过这种设置，HostV 不能直接从 Internet 访问，即不能直接从 HostU 访问。
3. 随后创建 TUN 虚拟网络接口，当程序从 TUN 接口读取数据时，计算机发送到此接口的 IP 数据包将被传送给程序，程序可以使用标准的 `read()` 和 `write()` 系统调用来接收或发送数据包到虚拟接口。
4. 最后再配置相对应的路由，当发送数据时根据相应的路由进行转发。

#### 任务二：实现隧道的加密

我们已经创建了一个 IP 隧道，但是我们的隧道没有受到保护。只有在我们保障了隧道的安全之后，才能将其称为 VPN 隧道。这就是我们在这项任务中要实现的目标。为了保护这条隧道，我们需要实现两个目标，即机密性和完整性。使用加密来实现机密性，即通过隧道的内容将被加密。完整性目标确保没有人可以篡改隧道中的流量或发起重放攻击。使用消息验证代码（MAC）可以实现完整性。可以使用传输层协议（TLS）实现这两个目标。

---

### 任务三：认证 VPN 服务器

在建立 VPN 之前，VPN 客户端必须对 VPN 服务器进行身份认证，确保服务器不是假冒的服务器。

VPN 服务器需要首先从证书颁发机构（CA）（例如 Verisign）获取公钥证书。当客户端连接到 VPN 服务器时，服务器将使用证书来证明它是客户端预期的服务器。Web 中的 HTTPS 协议使用这种方式来认证 Web 服务器，确保客户端正在与预期的 Web 服务器通信，而不是伪造的 Web 服务器。

### 任务四：认证 VPN 客户端

在建立 VPN 之前，VPN 服务器必须认证客户端（即用户），确保用户具有访问专用网络的权限。当用户尝试与 VPN 服务器建立 VPN 隧道时，将要求用户提供用户名和密码。服务器将检查其影子文件（/etc/shadow）；如果找到匹配的记录，则对用户进行认证，并建立 VPN 隧道。如果没有匹配，服务器将断开与用户的连接，因此不会建立隧道。

### 任务五：支持多客户端

真实应用中，一个 VPN 服务器通常支持多个 VPN 隧道。也就是说，VPN 服务器允许多个客户端同时连接到它，每个客户端都有自己的 VPN 隧道（从而有自己的 TLS 会话）。MiniVPN 应该支持多个客户端。

## 4 实验步骤及结果分析

### 4.1 建立主机到主机的隧道

#### 1. 配置环境

在计算机（客户端）和网关之间创建 VPN 隧道，允许计算机通过网关安全地访问专用网络。使用虚拟机本身作为 VPN 服务器网关（VM），并创建两个容器分别作为 VPN 客户端（HostU）和专用网络中的主机（HostV）。网络设置如图 4.1 所示：

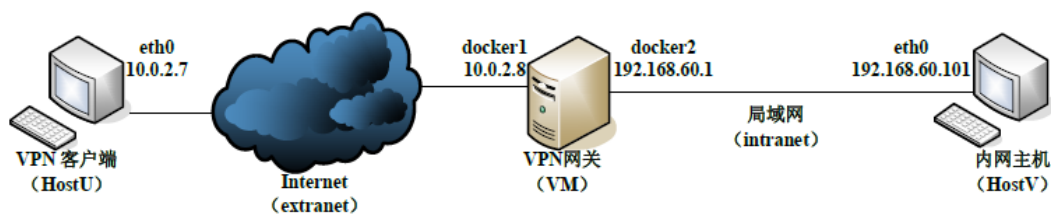


图 4.1 网络设置图

VPN 客户端 (HostU) 和 VPN 服务器网关的外网口通过 Internet 连接，在实验中将这两台机器直接连接到同一 docker 网络“extranet”，模拟 Internet。HostV 是内部局域网的计算机。Internet 主机 HostU 上的用户希望通过 VPN 隧道与内部局域网的主机 HostV 通信。为模拟此设置，我们使用 docker 网络“intranet”将 HostV 与 VPN 服务器网关的内网口连接，模拟内部局域网。

在 VM 上创建 docker 网络 intranet 和 extranet，使用指令如下，执行后的路由表中出现了相应的接口和网关如图 4.2 所示：

在 VM 上创建 docker 网络 extranet

```
$ sudo docker network create --subnet=10.0.2.0/24 --gateway=10.0.2.8 --opt
"com.docker.network.bridge.name"="docker1" extranet
```

在 VM 上创建 docker 网络 intranet

```
$ sudo docker network create --subnet=192.168.60.0/24 --gateway=192.168.60.1
--opt "com.docker.network.bridge.name"="docker2" intranet
```

```
root@VM:/home/seed/Desktop/TLS# route
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
default 192.168.230.2 0.0.0.0 UG 100 0 0 ens33
10.0.2.0 * 255.255.255.0 U 0 0 0 docker1
link-local * 255.255.0.0 U 1000 0 0 ens33
172.17.0.0 * 255.255.0.0 U 0 0 0 docker0
192.168.53.2 * 255.255.255.254 U 0 0 0 tun0
192.168.60.0 * 255.255.255.0 U 0 0 0 docker2
192.168.230.0 * 255.255.255.0 U 100 0 0 ens33
```

图 4.2 路由表

在 VM 上新开一个终端，创建并运行容器 HostU，同样的方式创建 HostV，使用 docker ps 查看创建的容器：

```
root@VM:/home/seed/Desktop/TLS# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d4755bd0e073 seedubuntu "/bin/bash" 9 days ago Up 7 days HostV
3f03239faf98 seedubuntu "/bin/bash" 9 days ago Up 7 days HostU
```

图 4.3 docker 创建

避免默认路由的干扰，在容器 HostU 和 HostV 内分别删除掉默认路由，如图 4.3 所示：

```
root@HostU:/# route del default
root@HostU:/# 
root@VM:/home/seed# sudo docker run -it --name=HostV --hostname=192.168.60.101 --privileged "seedubuntu" /bin/bash
root@HostV:/# route del default
root@HostV:/#
```

图 4.3 删除默认路由

## 2. 使用 TUN 技术创建隧道

用户空间程序通常访问 TUN 虚拟网络接口。操作系统通过 TUN 网络接口将数据包传送到用户空间程序。另一方面，程序通过 TUN 网络接口发送的数据包被注入操作系统网络栈，在操作系统看来，数据包是通过虚拟网络接口的外部源进来的。程序可以使用标准的 `read()` 和 `write()` 系统调用来接收或发送数据包到虚拟接口。隧道的拓扑如图所示：

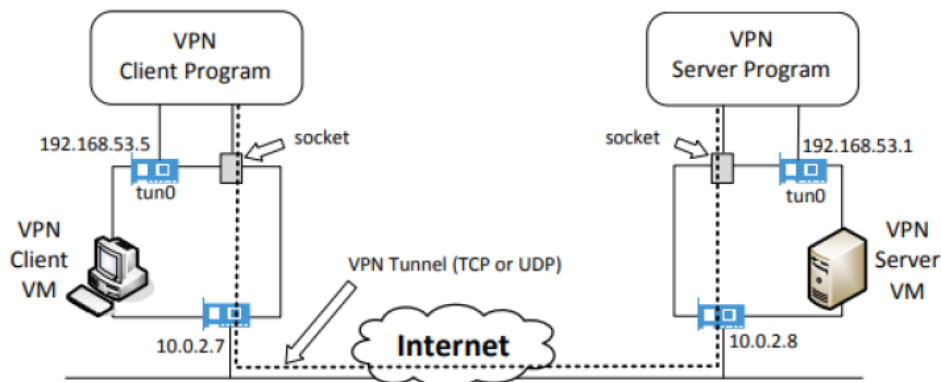


图 4.4 隧道连接拓扑

在 VM 上运行 VPN 服务程序，在程序的 `createTunDevice()` 函数中使用 `tunfd = open("/dev/net/tun", O_RDWR)`，打开了 tun 设备，同时 tun 的编号是系统内核自动顺序分配的，在后续实现多客户端时创建多 tun 实现中因为没注意这出现过问题。创建 tun 的具体代码如下：



```

memset(&ifr, 0, sizeof(ifr));

/* Flags : IFF_TUN   - TUN设备
 *          IFF_TAP   - TAP设备
 *          IFF_NO_PI  - 不需要提供包的信息
 */
ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

tunfd = open("/dev/net/tun", O_RDWR);
if (tunfd == -1) {
    printf("Open /dev/net/tun failed! (%d: %s)\n", errno, strerror(errno));
    return -1;
}
ret = ioctl(tunfd, TUNSETIFF, &ifr);
if (ret == -1) {
    printf("Setup TUN interface by ioctl failed! (%d: %s)\n", errno, strerror(errno));
    return -1;
}

```

图 4.5 tun 创建

在启动 server 程序后，tun 已经被创建，正在等待客户端的连接，但这个时候即使运行了客户端建立连接，也无法完成通信。因为 tun 没有被激活，路由也没有配置。

使用指令 `sudo ifconfig tun0 192.168.53.2/31 up` 来激活在 server 程序中创建的虚拟网卡并配置虚拟 IP：

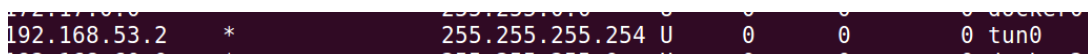


图 4.6 虚拟网卡的路由

除非特别配置，否则计算机将仅充当主机，而不充当网关。VPN Server 需要在内网和隧道之间转发报文，因此需要作为网关。我们需要为计算机启用 IP 转发，使其行为类似于网关。由于 VM 上的 iptables 规则可能阻断转发报文，我们还需要清除 iptables 规则。指令如下：

```

sudo sysctl net.ipv4.ip_forward=1

sudo iptables -F

```

在主机 HostU 上运行的客户端程序创建了 tun，也需要对该 tun 分配虚拟 IP 同时激活。但是仅仅完成和服务端一样的配置是不能通过隧道进行通信的，因为没有从主机转发到 tun 虚拟 IP 的转发路由。在 HostU 上，我们需要将所有进入专用网络（192.168.60.0/24）的数据包定向到 tun 接口，从该接口可以通过 VPN 隧道转发数据包。如果没有此设置，我们将无法访问专用网络。我们

可以使用 `route` 命令添加路由条目。使用指令 `route add -net 192.168.60.0/24 tun0` 后的路由表：

```
root@HostU: /# route
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
default          wujintianServer 0.0.0.0         UG    0      0      0 eth0
10.0.2.0         *               255.255.255.0   U      0      0      0 eth0
192.168.53.2     *               255.255.255.254 U      0      0      0 tun0
192.168.60.0     *               255.255.255.0   U      0      0      0 tun0
root@HostU: /#
```

图 4.7 HostU 的路由

这样便完成了隧道的建立同时数据可以在隧道上传输，但是外网主机却仍旧无法和内网主机 HostV 通信，因为 HostV 主机的回复报文没有转发路由，转发到服务器端进行加密，而后再经过隧道传输。

于是添加目标地址为外网虚拟 IP 网段的数据转发到 docker2 的路由，添加指令和结果如下：

```
root@HostV: /# route add -net 192.168.53.0/24 gw 192.168.60.1
root@HostV: /# route
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
192.168.53.0     192.168.60.1    255.255.255.0   UG    0      0      0 eth0
192.168.60.0     *               255.255.255.0   U      0      0      0 eth0
root@HostV: /#
```

图 4.8 HostV 的路由

### 3. 测试 VPN 隧道

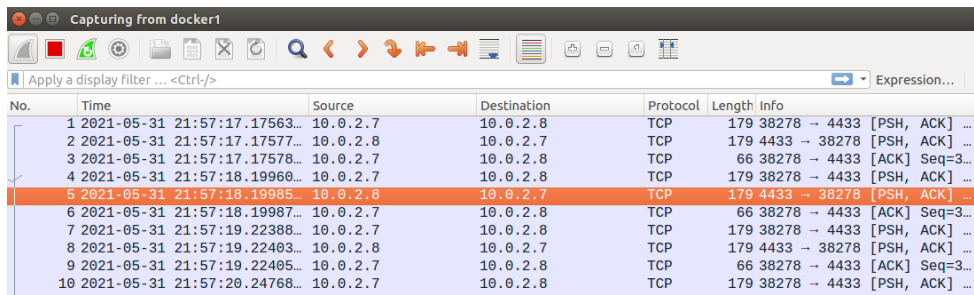
完成好配置后可以使用 `ping` 和 `telnet` 指令测试是否有数据在隧道上传输，可以通过 `wireshark` 查看隧道上的数据。隧道的两端分别为客户端的网关和服务器的网关，10.0.2.7 和 10.0.2.8。

使用 `ping` 指令 `ping` 内网主机 HostV 测试，外网主机 HostU 收到了回复报文，说明内外网之间可以进行通信：

```
root@HostU: /
192.168.60.0 * 255.255.255.0 U 0 0 0 tun0
root@HostU: /# ping 192.168.60.101
PING 192.168.60.101 (192.168.60.101) 56(84) bytes of data.
64 bytes from 192.168.60.101: icmp_seq=1 ttl=63 time=0.317 ms
64 bytes from 192.168.60.101: icmp_seq=2 ttl=63 time=0.208 ms
64 bytes from 192.168.60.101: icmp_seq=3 ttl=63 time=0.404 ms
64 bytes from 192.168.60.101: icmp_seq=4 ttl=63 time=0.233 ms
64 bytes from 192.168.60.101: icmp_seq=5 ttl=63 time=0.216 ms
64 bytes from 192.168.60.101: icmp_seq=6 ttl=63 time=0.233 ms
```

图 4.9 HostU 与 HostV 通信

然后在 wireshark 中的 docker1 端口，可以看到 10.0.2.7 和 10.0.2.8 之间的数据交互：(因为是已经完成加密的通信所以显示的是 TCP 报文而不是 ICMP)



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-05-31 21:57:17.17563...	10.0.2.7	10.0.2.8	TCP	179	38278 → 4433 [PSH, ACK] ...
2	2021-05-31 21:57:17.17577...	10.0.2.8	10.0.2.7	TCP	179	4433 → 38278 [PSH, ACK] ...
3	2021-05-31 21:57:17.17578...	10.0.2.7	10.0.2.8	TCP	66	38278 → 4433 [ACK] Seq=3...
4	2021-05-31 21:57:18.19960...	10.0.2.7	10.0.2.8	TCP	179	38278 → 4433 [PSH, ACK] ...
5	2021-05-31 21:57:18.19985...	10.0.2.8	10.0.2.7	TCP	179	4433 → 38278 [PSH, ACK] ...
6	2021-05-31 21:57:18.19987...	10.0.2.7	10.0.2.8	TCP	66	38278 → 4433 [ACK] Seq=3...
7	2021-05-31 21:57:19.22388...	10.0.2.7	10.0.2.8	TCP	179	38278 → 4433 [PSH, ACK] ...
8	2021-05-31 21:57:19.22403...	10.0.2.8	10.0.2.7	TCP	179	4433 → 38278 [PSH, ACK] ...
9	2021-05-31 21:57:19.22405...	10.0.2.7	10.0.2.8	TCP	66	38278 → 4433 [ACK] Seq=3...
10	2021-05-31 21:57:20.24768...	10.0.2.7	10.0.2.8	TCP	179	38278 → 4433 [PSH, ACK] ...

图 4.10 隧道流量

使用 telnet 进行的测试放在思考题中进行展示和说明。

## 4.2 隧道加密

为了保护这条隧道，我们需要实现两个目标，即机密性和完整性。使用加密来实现机密性，即通过隧道的内容将被加密。完整性目标确保没有人可以篡改隧道中的流量或发起重放攻击。使用消息验证代码(MAC)可以实现完整性。

为了实现这两个目标，在服务端和客户端的程序中使用传输层协议(TLS)实现这两个目标。实现加密的主要流程：

第一步是使用初始化 OpenSSL 库，使用相关的函数加载 ssl 上下文 ctx，再制定证书验证方法，加载根证书，最后使用 SSL\_new(ctx)生成一个 SSL\*类型变量。

第二步是创建一个服务器端和客户端的 TCP 套接字，生成一个 sockfd。TCP 套接字的创建流程如下图：

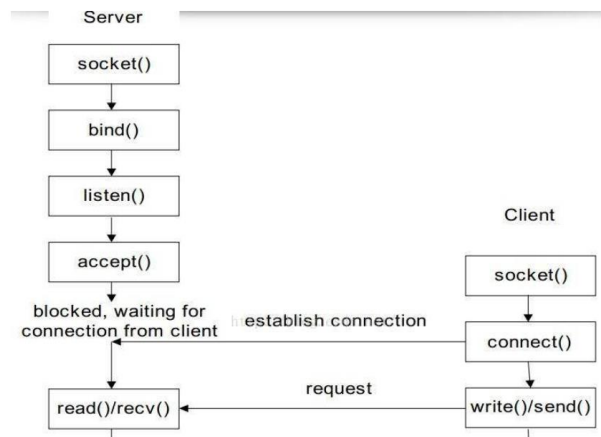


图 4.11 tcp 套接字创建

最后一步就是使用 `SSL_set_fd(ssl, sockfd);` 将 `ssl` 和相应的套接字连接，在客户端使用 `SSL_connect()`，在服务端使用 `SSL_accept()` 进行 TLS 握手协议。相关流程如下图所示：

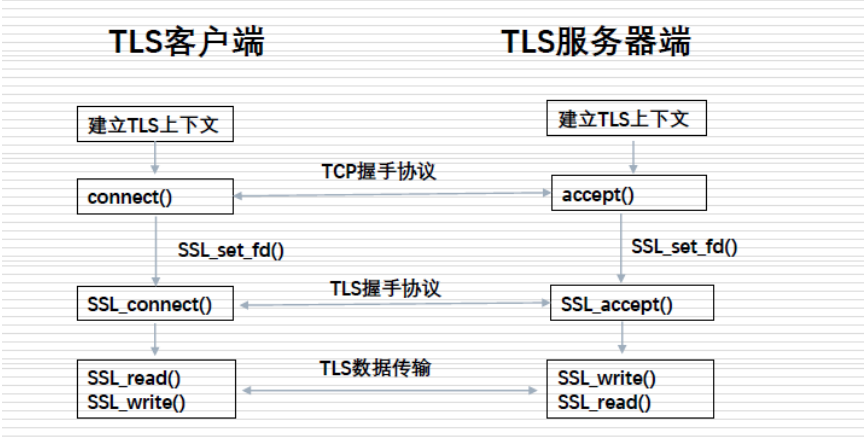


图 4.12 TLS 连接

完成这些后，就可以使用 `tunSelected()`、`socketSelected()` 函数进行监测，而后使用 `SSL_write` 和 `SSL_read` 函数进行服务端和客户端的通信。

加密隧道的检验，查看 `docker1` 端口的 `ping` 报文数据是否为加密数据：

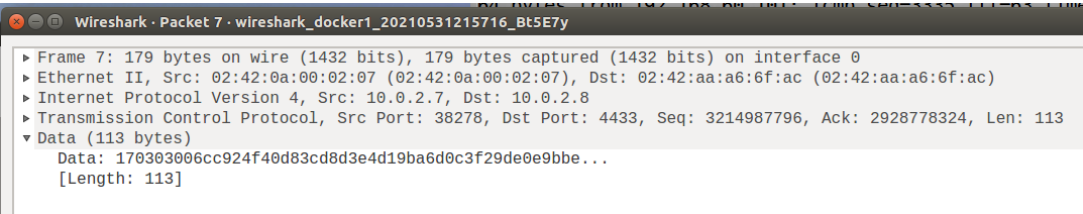


图 4.13 加密 ping 报文数据

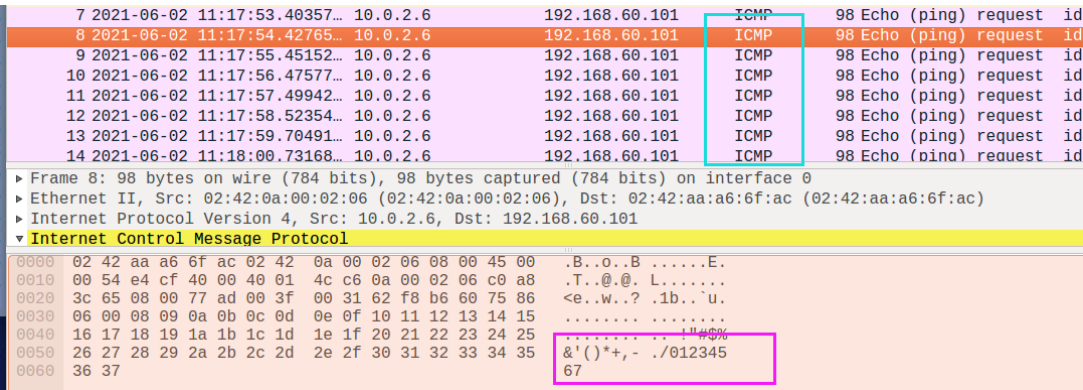


图 4.14 未加密 ping 报文

未加密的 ping 报文数据是 123456 如图 4.14 所示，而此数据并不是，同时 wireshark 将该报文识别为 TCP 报文而非 ICMP 报文，所以加密隧道的加密成功。同时可以在运行程序的输出中查看到数据在 tun 和 tunnel 之间的流动：

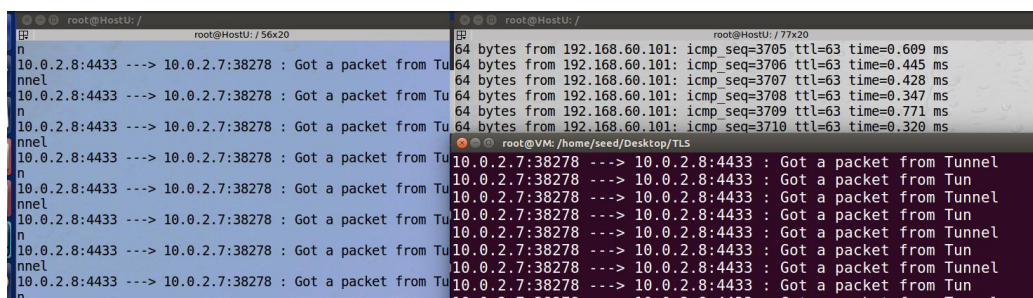


图 4.15 程序打印信息

### 4.3 认证 VPN 服务器

在建立 VPN 之前，VPN 客户端必须对 VPN 服务器进行身份认证，确保服务器不是假冒的服务器。

在实验中对服务器的认证是使用公钥证书实现的。VPN 服务器需要首先从证书颁发机构（CA）获取公钥证书。当客户端连接到 VPN 服务器时，服务器将使用证书来证明它是客户端预期的服务器。

使用 openssl 相关的指令生成 CA、服务器证书、客户端证书，将证书文件夹复制到 HostU 中。在生成证书的过程中需要注意的地方：

1. 要根据 openssl.cnf 文件设置相应的目录文件
2. 创建的 index.txt 文件为空文件
3. 创建的 serial 文件中写入 1000

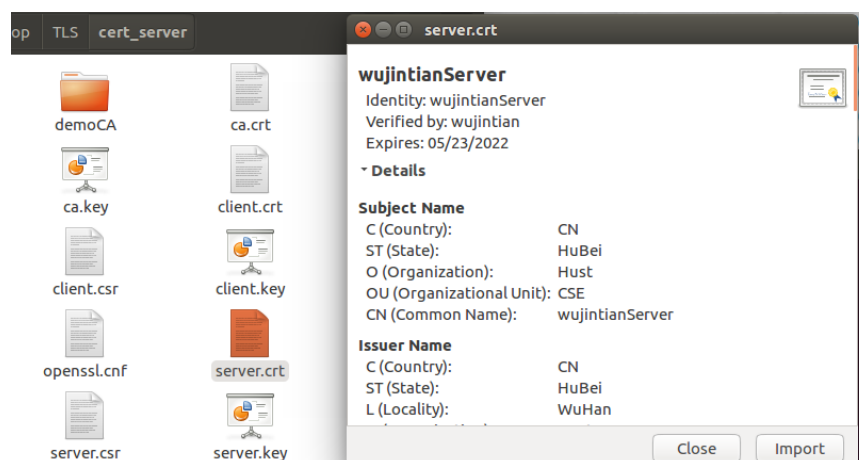


图 4.16 证书文件目录



生成了服务器的相关证书后，还需要在程序中将证书的信息加载在一次通话的 ssl 中，让客户端使用根 CA 来验证服务端的加载在通话 ssl 中的证书是否合法。以下解释一下在程序中使用的一些 SSL 函数：

为了使证书正确的加载，在程序中定义了各证书文件的路径以及文件名：

```
#define BUFF_SIZE 2000
/* define HOME to be dir for key and cert files... */
#define HOME      "./cert_server/"

/* Make these what you want for cert & key files */
#define CERTF     HOME"server.crt"
#define KEYF      HOME"server.key"
#define CACERT    HOME"ca.crt"
```

图 4.17 文件名

在服务端程序中先使用 `SSL_CTX_set_verify()` 函数，函数的第二个参数使用 `SSL_VERIFY_NONE` 参数表示不进行验证，而在客户端的程序中使用 `SSL_VERIFY_PEER` 参数表示对对等方的证书进行验证。在函数中使用 `SSL_CTX_load_verify_locations()` 函数将根 CA 加载在上下文中，客户端加载 CA 进行验证服务器端证书的合法性。

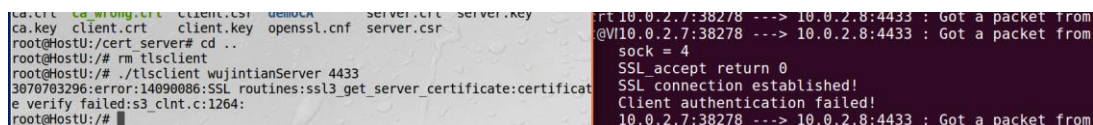
```
// Step 1: SSL context initialization
meth = SSLv23_server_method();
ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, verify_callback);
//SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, CACERT, NULL);
```

图 4.18 server 端证书加载

```
//制定证书验证方式的函数
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, CACERT, NULL);
```

图 4.19 client 端验证方式

随后运行两端程序即可。验证是否真正的进行了验证，在客户端使用伪造的 `server.crt`，而后运行程序，会有错误提示，如图 4.19 所示：



```
ca.crt ca.key client.crt client.key openssl.cnf server.crt server.key
root@HostU:/cert_server# cd ..
root@HostU:/# rm tlsclient
root@HostU:/# ./tlsclient wujintianServer 4433
3070703296:error:14090086:SSL routines:ssl3_get_server_certificate:certificate verify failed:s3_clnt.c:1264:
root@HostU:/#
```

图 4.20 服务器证书验证

在实验过程中除开证书生成的错误，还有几个错误值得记录和注意：

1. 在编写服务端程序时，需要使用私钥文件 `server.key`，使用该文件需要输入密码。而当程序中添加 `daemon(1,1)` 时，会将执行的服务器程序放到后台运行，而后将控制权转交给 `bash`。如果将 `daemon(1,1)` 函数放在 `SSL_CTX_use_PrivateKey_file()` 函数之前，那么当输入私钥文件密码的时候，终端将密码识别为 `linux` 的控制台指令，同时还有回显。这样之后，甚至整个终端陷入卡死状态。

改错方法：只需要将 `daemon(1,1)` 放在输入之后即可。

2. 运行客户端程序时输入服务器的 IP 不能正确执行，需要在程序中编写根据主机名获取 IP 地址的函数。在执行客户端程序时将服务器域名作为参数输入。在程序中主要使用 `getaddrinfo()` 函数获取 IP 地址。同时需要在 `/etc/hosts` 文件中添加服务器域名对应的 IP。

```
root@HostU:/# cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0     ip6-localnet
ff00::0     ip6-mcastprefix
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
10.0.2.7    HostU
10.0.2.8    wujintianServer
root@HostU:/#
```

图 4.21 /etc/hosts 文件

#### 4.4 认证客户端

对于客户端的认证，使用的是存储在 `shadow` 文件中的帐户信息对用户进行身认证。该程序使用 `getspnam()` 从 `shadow` 文件中获取给定用户的帐户信息，包括散列密码。然后，它使用 `crypt()` 来散列给定的密码，并查看结果是否与从 `shadow` 文件中获取的值匹配。如果是，则用户名和密码匹配，并且验证成功。

在服务端实现这个函数是很简单的，函数代码如下所示：

```
int login(char* user, char* passwd)
{
    struct spwd* pw;
    char* epasswd;
    pw = (struct spwd*)getspnam(user);
    if (pw == NULL) {
        return 0;
    }
    printf("Login name: %s\n", pw->sp_namp);
    printf("Encrypted Passwd : %s\n", pw->sp_pwdp);
    epasswd = crypt(passwd, pw->sp_pwdp);
    if (strcmp(epasswd, pw->sp_pwdp)) {
        return 0;
    }
    return 1;
}
```

图 4.22 login 代码

如上图代码所示，当输入密码和用户名匹配时，返回数字 1；错误或者不匹配时返回 0。在服务器端对返回值的判断进行相应的操作，当验证成功时服务器端使用 `SSL_write()`向客户端发送数据“Yes”，否则发送“No”并且断开该次连接。

```
if (login(username, passwd)) { //返回为1 那么说明验证成功
    printf("Verified successfully!\n");
    //create the tun
    int tunfd;
    tunfd = createTunDevice((ip - 2) / 2);

    char buff[100];
    strcpy(buff, "Yes");
    buff[3] = '\0';
    SSL_write(ssl, buff, 3);

    //认证连接的时候不会经过虚拟网卡 直接通过s:
    //服务端分配虚拟网卡ip 发送给客户端 客户端生

} else { //send fail packet to client and close the ssl
    char buff[100];
    buff[0] = 'N';
    buff[1] = 'o';
    buff[2] = '\0';
    SSL_write(ssl, buff, 2);
    printf("Client authentication failed!\n");
    int r = SSL_shutdown(ssl);

    //retrieve the ip
    flag[ip] = flag[ip + 1] = 0;
    //error handling here if r < 0
}
```

在服务器端对客户端进行验证，客户端自然需要将用户名和密码传递给服务器，通过简单的 `scanf` 就能获取输入，而后使用 `SSL_write` 将数据发送给服务器即可。

在实验中使用了 `termios` 结构体关闭回显，让输入客户密码时密码不被显示。具体代码如下：

```
int getpasswd(char* dest)
{
    struct termios oldflags, newflags;
    //设置终端为不回显模式
    tcgetattr(fileno(stdin), &oldflags); //fileno(stdin)获得标准输入的文件描述符
    newflags = oldflags;
    newflags.c_lflag &= ~ECHO; //ECHO 表示显示输入字符 ~ECHO就是不显示咯
    newflags.c_lflag |= ECHONL; //如果ICANON同时设置，即使ECHO没有设置依然显示换行符
    if (tcsetattr(fileno(stdin), TCSANOW, &newflags) != 0) //设置stdin 为 newflags
    {
        perror("tcsetattr");
        return -1;
    }
    //获取来自键盘的输入
    scanf("%s", dest);

    //恢复原来的终端设置
    if (tcsetattr(fileno(stdin), TCSANOW, &oldflags) != 0)
    {
        perror("tcsetattr");
        return -1;
    }
    return 0;
}
```

图 4.23 关闭密码的回显

在 `getpasswd()`函数中，设置标准输入文件描述符不显示输入字符，但显示换行符，在完成输入后，恢复初始的终端设置。以下是客户端输入密码的截图：



```

root@HostU:/# ./tlsclient wujintianServer 4433
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Please enter your Username and Password
UserName:seed
Password:
Verified successfully!
root@HostU:/# Setup TUN interface success!
10.0.2.8:4433 ---> 10.0.2.7:38366 : Got a packet from T
n

root@VM:/home/seed/Desktop/TLS# sock
SSL_accept return 1
SSL connection established!
Login name: seed
Encrypted Passwd : $6$wDRrWCQz$IsBXp9
sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/
Verified successfully!
Setup TUN0 interface success!
分配服务端的 tun0 的 IP 为 : 192.168.53.2
分配客户端的 IP 为 : 192.168.53.3

```

图 4.24 关闭回显

验证当客户端输入错误密码时:

```

root@HostU:/# ./tlsclient wujintianServer 4433
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Please enter your Username and Password
UserName:dees
Password:
Client authentication failed!root@HostU:/#

root@VM:/home/seed/Desktop/TLS# ./tlsserver
Enter PEM pass phrase:
root@VM:/home/seed/Desktop/TLS# listen_sock = 3
sock = 4
SSL_accept return 1
SSL connection established!
Client authentication failed!

```

图 4.25 输入错误密码

## 4.5 多客户端实现

在现实情况中通常会有多个客户端同时连接服务器，同时进行数据传输，在实验中也实现了这个功能。

实现该功能的难点在于需要给客户端的 tun 分配虚拟 IP，同时需要知道内网发送的数据要经过哪条隧道发送到目标主机。为了解决这两个问题，在服务端验证完客户端后，每一个连接都为它创建一个 tunX(X 顺序递加)，同时为对应的客户端分配虚拟 IP。

以下展示实现的代码流程以及讲解：

1. 为建立连接的服务端和客户端获取可用的虚拟 IP：

在程序中将一个连接的虚拟 IP 分为一组划分为网络前缀为 31，例如：设置让服务端的 ip 为 192.168.53.2 、 4 、 6 、 8 等等，对应客户端的虚拟 IP 分别为 3，5，7，9 等。

在服务器程序的 accept()函数之后，通过对 flag[256]数组的扫描，判断哪个 IP 未被使用，则准备将这个 IP 分配给服务端的虚拟网卡 tunX( $X=(IP-2)/2$ )，将 IP+1 分配给对应的客户端。

```

unsigned int ip;
//get an available ip for connection
for (int i = 1; i < 127; i++) {
    if (!flag[2 * i]) { //available
        ip = 2 * i; //ip for server
        //client ip is ip+1
        flag[2 * i] = flag[2 * i + 1] = 1; //used
        break;
    }
}

```

图 4.26 get the ip

## 2. 将获取的可用虚拟 IP 进行使用:

在 1 中获取了一组可用的虚拟 IP，如果通过了两端的认证，则服务器端根据获取的可用虚拟 IP 创建对应的 tunX(X=(IP-2)/2)，同时在程序中构建字符串指令，利用 system()函数激活 tunX，以下为相关代码：

双方均验证成功后，将 IP 分配给对应的客户端，通过 SSL\_write()函数直接告诉对端：

```

//认证连接的时候还不会经过虚拟网卡 直接通过ssl层交互
//服务端分配虚拟网卡ip 发送给客户端 客户端生成对应的虚拟网卡ip 并添加相应的路由
unsigned int Cip = ip + 1;
memset(buff, 0, sizeof(buff));
memcpy(buff, &Cip, sizeof(int));
SSL_write(ssl, buff, 4);
printf("\033[22;31m分配服务端的tun%d的IP为：192.168.53.%d\n", (ip - 2) / 2, ip);
printf("分配客户端虚拟IP：192.168.53.%d\n", Cip); //输出红色文字
printf("\033[22;39m"); //恢复黑色

```

图 4.27 分配虚拟 IP

在 createTunDevice()函数中构建激活 tunX 的指令，下图中的 my\_itoa()函数是自己编写的 itoa 函数，即将整数转化为字符串：

```

//构建指令
char instr[100], instruction2[100];
memset(instr, 0, sizeof(instr));
memset(instruction2, 0, sizeof(instruction2));
strcpy(instr, "sudo ifconfig tun"); //set up the tun
//strcpy(instruction2, "route add -net 192.168.60.0/24 tun"); //add the routes
int len = strlen(instr);
//int len2 = strlen(instruction2);
char tmp[10];
memset(tmp, 0, sizeof(tmp));
my_itoa(ip, tmp); //将无符号整数转化为字符串
for (int i = 0; i < strlen(tmp); i++) {
    instr[len++] = tmp[i];
    //instruction2[len2++] = tmp[i];
}
strcat(instr, " 192.168.53.");
len = strlen(instr);
memset(tmp, 0, sizeof(tmp));
my_itoa(Sip, tmp);
for (int i = 0; i < strlen(tmp); i++) {
    instr[len++] = tmp[i];
}
strcat(instr, "/31 up");

```

图 4.28 激活服务端 tunX

---

在客户端经过 SSL\_read 读出分配给自己的虚拟 IP 后, 在 createTunDevice() 函数中创建该虚拟 IP 的 tun0, 激活后还需要添加一条转发路由:

```
//路由转发
const char* instruction2 = "route add -net 192.168.60.0/24 tun0";
system(instr);
system(instruction2);
```

图 4.29 添加路由

如果双方没有验证成功, 则需要将获取的虚拟 IP 进行回收:

```
//retrieve the ip
flag[ip] = flag[ip + 1] = 0;
```

图 4.30 ip 回收

3. 完成虚拟 IP 分配后, 客户端可以和服务器端通信:

使用服务器端一个虚拟网卡对应一个客户端的好处在于可以直接的进行通信, 和单客户端的读写流程一致。使用管道进行通信需要建立对管道的监听, 将管道的数据进行针对性的转发。

4. 虚拟 IP 的回收:

在进行虚拟 IP 的回收时利用了一个父子进程的常用函数 exit()、以及 waitpid(), 当客户端结束通信时, 通过 SSL\_write()将虚拟 IP 告诉给服务器对应的子进程, 子进程也通过 exit(Cip)退出, 同时将 Cip 传递给父进程。父进程通过 waitpid(-1,&status,WNOHANG), 接收 Cip, 并将 flag[Cip-1]和 flag[Cip]置为 0, 表示为可用。

在 accept()函数后调用 signal()函数, 表示有新连接建立后执行 signal 中的回调函数:

```
// deal with defunct process
signal(SIGCHLD, &func_waitpid);
```

图 4.31 signal 函数对子进程处理





随后断开 HostW 的连接,查看到对应的虚拟 IP 组:192.168.53.4/31 被回收,由于在程序中使用了 waitpid()接受服务端子进程的退出状态故不会使得生成的子进程为僵尸进程。如下图 4.33 所示,结束了一个客户端进程,对应连接的服务端子进程也随之结束,此时服务端仅有一个主进程等待新的连接建立和一个已建立连接的子进程服务:

图 4.35 退出客户端 IP 回收

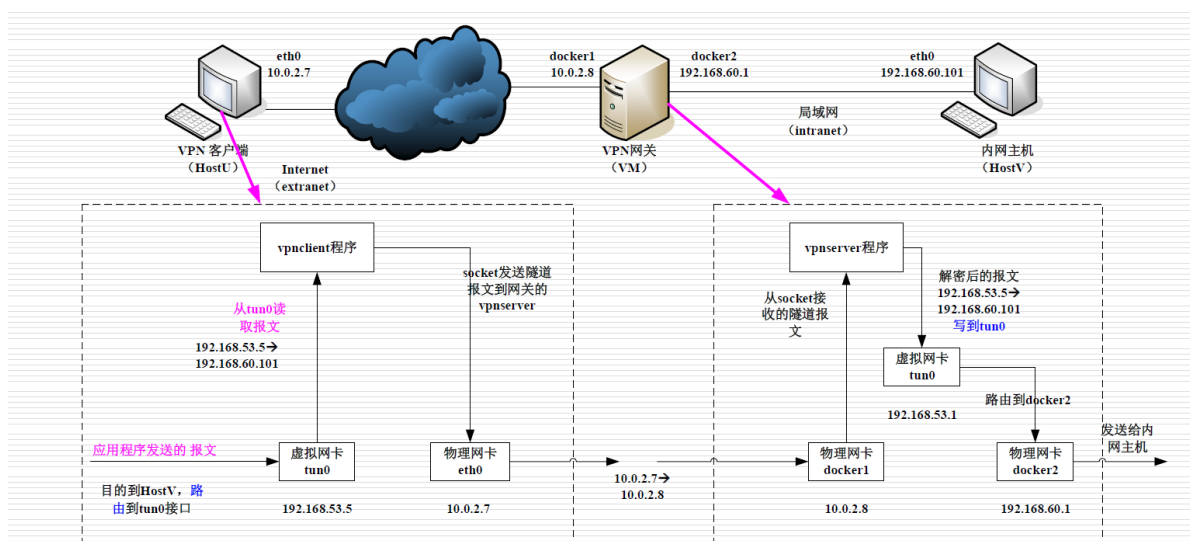
当主机 HostW 再次建立连接的时候,重新分配的 IP 就是最小的可用 IP:

图 4.36 重新建立连接分配的虚拟 IP

由上图可知,当有用户退出,该用户使用的虚拟 IP 被回收了,另一个用户连接时可以获得该分配的 IP。

## 4.6 报文处理流程

以下讲解一次客户端与服务器端交互的报文处理,以下为网络拓扑图:



---

图 4.37 网络拓扑

1. 外网主机发出数据
2. 数据需要经过虚拟网卡 tun0 发送数据，从 tun0 发送出去之前，内核使用虚拟网卡的虚拟 IP 作为源 IP 地址来构造程序数据的 IP 数据包。此后虚拟网卡 tun0 将源 IP 为:192.168.53.5，目标地址为 192.168.60.101 的 IP 数据包根据路由表中的路由将数据包转发给 VPN 客户端程序。
3. IP 数据包通过虚拟接口 tun0 来到了我们的 VPN 客户端程序，之后数据包会被加密再发送给内核，这次将客户端的网关和服务端的网关作为通信隧道，即此时封装的 IP 数据包源 IP:10.0.2.7，目的 IP: 10.0.2.8。
4. 此时加密数据就在 Internet 中传递转发，即可保障数据在传输过程的机密性。
5. 数据包到达服务器端的网关后完成了一步解封装，到达服务器程序进行解密，然后再通过服务器端的虚拟网卡将解密后的程序数据包传递给内核。这时的数据包源 IP: 192.168.53.5，目的 IP: 192.168.60.101。
6. 内核查看目标地址，并根据它进行路由转发到对应的内网主机。最终数据包达到 192.168.60.101 主机。

内网主机的回复过程也类似，只是 IP 不同而已，流程和封装过程是一样的。在保持 telnet 连接存活的同时，断开 VPN 隧道。

## 5 实验思考

### 测试:

当 HostU 主机使用 telnet 连接上 HostV 后，再断开 VPN 隧道，HostU 主机的 telnet 终端界面卡死了，无法输入任何的指令。

重新连接 VPN 隧道时，需要将 vpnserver 和 vpnclient 都退出后再重复操作，因为 tun 可以认为是配置在程序中的软件，断开隧道后 tun 虚拟网卡接口也断开了，需要重新配置激活后再进行连接。当正确重连后，可以继续通信了，之前卡住的 HostU 主机的终端界面可以输入指令并成功执行。

## 思考与分析：

在隧道未断开的时候，当客户端与服务器互相验证通过后，在客户机使用 telnet 指令和内网主机建立连接。HostU 和内网主机已经建立了 telnet 连接，这时候断开 VPN 隧道，HostU 和内网主机之间的连接仍然存在着，只是没有了传输数据的路径（就是隧道）。当 VPN 隧道重新连接的时候，HostU 和内网主机之间的通信要求 1.两者之间建立者连接 2.有着数据传输路径，这两条都满足了，自然可以继续通信了。

而且发现，重新建立连接后的 docker2 中的 telnet 会话的 ACK 是和断开前最后一个报文的 Seq 对应的，这也说明了其实隧道断开但是会话并没有断开。重新建立隧道后就可以继续通信。



图 5.1 会话 ACK

---

## 心得体会与建议

### 1 心得体会

本次实验确实收获了不少，在刚开始的时候对照着老师所给的 `vpn` 程序和 `tls` 程序一个函数一个函数的对照着看函数的参数以及函数的功能。在学习和查找资料的过程中，又复习了一下上个学期使用过的 `socket` 套接字，同时对于 `TCP` 建立连接时实现也有了更多的理解。基于 `TCP` 的 `socket` 编程，在服务器端程序的流程：

1. 创建套接字 `socket`
2. 再使用 `bind` 函数将套接字绑定到本地地址和端口
3. 使用 `listen` 函数监听客户端的连接请求，将请求加入到队列中
4. 使用 `accept` 函数从监听队列中取出一个请求进行连接并返回一个会话套接字
5. 使用该会话套接字进行通信使用 `send` 和 `recv` 函数，在程序中使用的是 `SSL` 的 `SSL_write` 和 `SSL_read` 函数
6. 关闭会话套接字

我觉得实验的难点在于对一些函数的底层实现和原理不太了解而产生的错误，以及多客户端的实现。

在实现双方身份认证的时候，将 `daemon` 函数放在了 `SSL_CTX_use_PrivateKey_file` 函数前，这将产生终端直接卡死的错误。使用了 `daemon` 函数将程序转到后台，而运行到 `SSL_CTX_use_PrivateKey_file` 函数是需要输入文件密码，但是此时控制权从程序转交回给 `bash`，输入的密码被识别为指令，同时终端卡死。

在实现多客户端功能时，一开始使用的是管道进行父子进程之间的通信，主进程将分配的虚拟 IP 通过管道发送给对应的子进程，子进程在 `selectPipe` 函数中获取自己对应管道写来的信息。但是在实现的过程中出现了问题，对于多客户端进行 `ping` 指令时，丢包率不为 0，怀疑是管道的管理出现问题，但是查



---

看了很多遍的代码没有看出问题在哪，于是改用了服务器创建多虚拟网卡来对应客户机的方法。

在使用多虚拟网卡的过程中，一开始是将虚拟 IP 的最后八位的数值当作 tun 的编号，但是运行时会提示 No such device 的错误。这表明在程序中创建的 tunX 并没有成功，但是使用 `ifconfig -a` 查看到所有创建的 tun 都是顺序编号的，这个编号其实在使用 `tunfd = open("/dev/net/tun", O_RDWR)` 的时候系统自动进行了分配。所以建立一个服务端虚拟 IP 到 tun 编号的线性映射，这样直接可以通过分配的虚拟 IP 来激活对应的虚拟网卡。

总的来说，这次的实验收获挺大的，不仅对 SSL VPN 的原理和实现有了更加深刻的了解，也重温了一下 socket 编程和多进程。

也非常感谢各位老师和助教的耐心解答！

## 2 建议

感觉大部分的代码都已经给了出来，实际完整编写的功能就是多客户端了，感觉可以不需要那么多的提示代码。