华 中 科 技 大 学

网络空间安全学院

本科：《计算机网络安全实验》报告

题目：TLS VPN 设计与实现

姓　　名　柳骁恩

班　　级　网安 2304 班

学　　号　U202314510

联系方式　13635763585

分　　数

评 分 人

2026 年 1 月 3 日

报告要求

1. 报告不可以抄袭，发现雷同者记为 **0** 分。

2. 报告中不可以只粘贴大段代码，应是文字与图、表结合的，需要说明流程的时候，也应该用流程图或者伪代码来说明；如果发现有大段代码粘贴者，报告需重写。

3. 报告格式严格按照要求规范，并作为评分标准。

报告评分表

评分项目	分值	评分标准	得分
实验原理	10	10-8：原理理解准确，说明清晰完整 7-5：原理理解基本准确，说明较为清楚 4-0：说明过于简单	
VPN 系统设计	25	25-19：系统架构和模块划分合理，详细设计说明翔实准确 18-11：系统架构和模块划分基本合理，详细设计说明较为准确 10-0：系统架构和模块划分不恰当，详细设计说明过于简单	
VPN 实现细节	25	25-19：回答准确清晰，实现方法技术优良，与设计及代码一致 18-11：回答较准确清晰，实现方法一般，与设计及代码有偏差 10-0：文字表达混乱，实现方法过于简单	
问题分析与解决	20	20-15：问题描述清晰，原因分析准确，解决方法正确合理 14-9：问题描述较清晰，原因分析比较准确，解决方法基本可行 8-0：问题描述含糊不清，原因不准确，解决方法不合理	
体会与建议	10	10-8：心得体会真实，意见中肯、建议合理可行，体现个人思考 7-5：心得体会较为真实，意见建议较为具体 4-0：过于简单敷衍	
格式规范	10	图、表的说明，行间距、缩进、目录等不规范相应扣分	
总 分			

目　　录

1 实验原理

1.1 网络拓扑

本实验将在计算机（客户端）和网关之间创建 TLS/SSL 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（VM），其在外网的 IP 地址为 10.0.2.8，在内网的 IP 地址为 192.168.60.1。并创建三个容器分别作为 VPN 客户端（HostU、HostU2）和专用网络中的主机（HostV）。为 HostU 分配 IP 地址为 10.0.2.7，为 HostU2 分配 IP 地址为 10.0.2.6，为 HostV 分配 IP 地址为 192.168.60.101。实验网络拓扑图如图 1-1 所示。

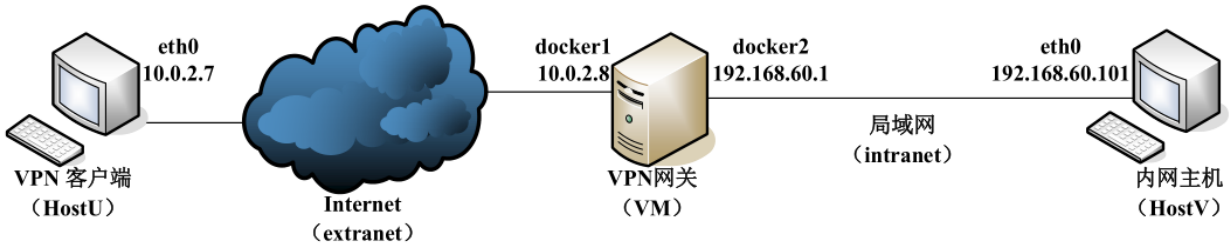

图 1-1 实验网络拓扑图

1.2 通信机制

本实验使用 Linux 内核提供的 TUN/TAP 虚拟网络设备实现隧道功能。其中 TUN 技术使虚拟网络内核驱动程序，用来模拟网络层设备，处理第三层 IP 数据包，允许用户直接与网络层或者数据链路层交互。VPN 客户端和服务端通过 TUN 接口建立隧道的通信流程如图 1-2 所示。

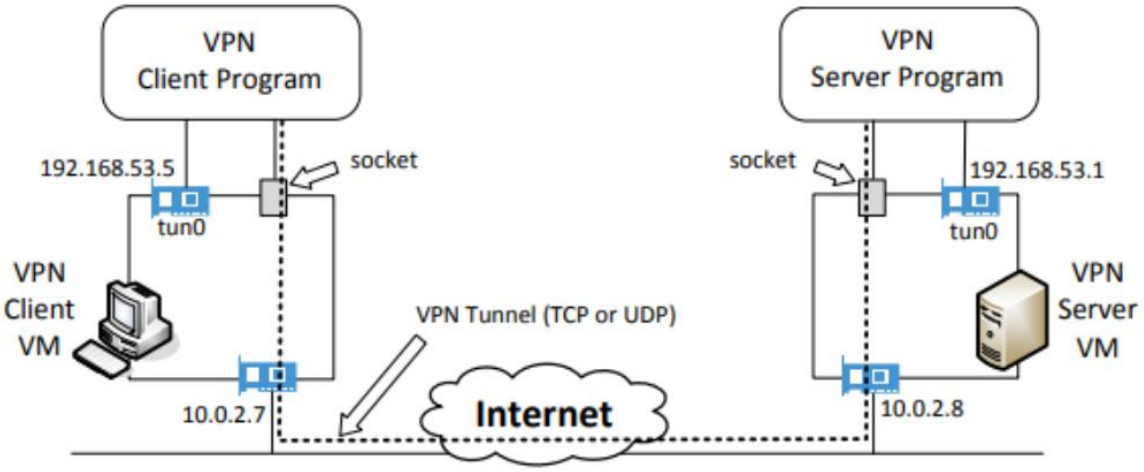

图 1-2 隧道示意图

以外网主机 HostU 向内网主机 HostV 发送数据包为例：

1.在 VPN 客户端（HostU） 和 VPN 服务器（VM） 上分别创建 tun0 虚拟接口，并分配同一子网内的 IP 地址（如 192.168.53.5/24 和 192.168.53.1/24）。

2.用户空间程序（vpnclient / vpnserver）通过 read()/write() 系统调用与 tun0 接口交互：

3.当 HostU 向 HostV（192.168.60.0/24）发送数据时，内核根据路由表将 IP 包送入 tun0；

4.vpnclient 从 tun0 读取该 IP 包，通过底层传输通道（初始为 UDP，后升级为 TCP+TLS）发送给 vpnserver；

5.vpnserver 收到后，将原始 IP 包写入其本地 tun0，由内核继续转发至 HostV。

1.3 加密原理

初始隧道仅提供连通性，不具备安全性。为满足 机密性 与 完整性 要求，实验将底层传输协议从 UDP 升级为 TCP + TLS/SSL。利用 OpenSSL 库建立 TLS 会话：

机密性：通过 TLS 握手协商出的对称会话密钥，对隧道内所有 IP 数据包进行加密（如 AES）。

完整性：TLS 使用 HMAC 或 AEAD 模式（如 AES-GCM）确保数据未被篡改或重放。

所有原始 IP 包作为 TLS 记录的载荷进行传输，外部观察者（如 Wireshark 抓包）仅能看到加密后的 TLS 流量，无法解析内部通信内容。

1.4 认证机制

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。另一方面，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。

认证服务器的典型方法是使用公钥证书。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。

对客户端的身份认证一般 VPN 其实是不包括的，但是可以通过登录账户和用户名进行认证。服务器会验证客户端的账户和密码是否和其内存中的记录情况匹配，如果有相应的记录，则可通过身份认证。

2 VPN 系统设计

2.1 概要设计

2.1.1 系统架构概述

本实验实现的 VPN 是一个基于 TLS/SSL 的虚拟专用网络系统，采用客户端-服务器架构。系统通过 TUN 虚拟网络接口实现 IP 层隧道，使用 TLS/SSL 协议加密通信内容，支持多客户端连接，并实现了服务器证书认证和客户端用户名密码认证。

系统整体架构如图 2-1、图 2-2 所示。

<table>
<tr><td colspan="3">VPN 客户端 （HostU）</td></tr>
<tr><td>应用层</td><td colspan="2">Telnet/Ping 等应用程序</td></tr>
<tr><td>系统层</td><td colspan="2">路由表 dest:192.168.60.0/24 -> tun0</td></tr>
<tr><td rowspan="2">VPN程序层</td><td>myclient.c</td><td>myclient.c</td></tr>
<tr><td>主线程
线程2</td><td>监听SSL，接收服务端数据->写入tun0
监听tun0，读取本地数据-> SSL加密发送</td></tr>
<tr><td>网络接口层</td><td colspan="2">tun0(192.168.53.5)　　eth0(10.0.2.7)</td></tr>
</table>

图 2-1 客户端整体架构图

<table>
<tr><td colspan="3">VPN 服务器 （VM）</td></tr>
<tr><td rowspan="3">VPN程序层</td><td colspan="2">myserver.c</td></tr>
<tr><td>父进程</td><td>1. 监听TCP连接，fork子进程
2. 监听tun0，通过管道转发到对应子进程
3. 管理所有客户端管道文件</td></tr>
<tr><td>子进程N</td><td>1. TLS握手、用户认证
2. 监听SSL和对应管道，双向转发数据</td></tr>
<tr><td>系统层</td><td colspan="2">IP转发：net.ipv4.ip_forward=1
路由：dest:192.168.53.0/24 → tun0
dest:192.168.60.0/24 → docker2</td></tr>
<tr><td>网络接口层</td><td colspan="2">tun0(192.168.53.1)　docker1(10.0.2.8)　docker2(....)</td></tr>
</table>

图 2-2 服务器整体架构图

2.1.2 模块划分

1. 服务器端模块（myserver.c）

（1）TLS/SSL 初始化模块

1）使用 OpenSSL 库初始化 TLS 上下文（SSL_CTX）。

2）加载服务器证书和私钥，设置不验证客户端证书（SSL_VERIFY_NONE）。

3）提供 setupTLSServer() 函数，用于创建并返回 SSL 结构体。

（2）TCP 服务器模块

1）创建监听套接字，绑定到指定端口（如 4433）。

2）使用 accept() 循环接收客户端连接。

3）为每个客户端创建子进程，实现多客户端支持。

（3）TUN 设备管理模块

1）调用 createTunDevice() 创建 TUN 虚拟接口。

2）配置 TUN 接口 IP 地址，并启用系统 IP 转发。

3）启动独立线程监听 TUN 接口，接收来自内网的数据包。

（4）客户端认证模块

1）通过 SSL 读取客户端发送的用户名和密码。

2）调用 login() 函数，使用 getspnam() 和 crypt() 验证 shadow 文件中的用户凭证。

3）认证成功后才建立隧道。

（5）进程间通信（IPC）与多隧道管理模块

1）为每个客户端创建一个命名管道文件，路径为 ./pipe/<客户端 IP 尾号>。

2）父进程监听 TUN 接口，根据数据包目标 IP 尾号选择对应管道转发数据。

3）子进程监听对应管道，将数据通过 TLS 发送给对应客户端。

4）使用 pthread 实现多线程并发处理。

（6）数据转发模块

1）子进程通过 SSL 读取客户端数据，写入 TUN 接口。

2）父进程读取 TUN 数据，通过管道转发至对应客户端子进程。

2. 客户端模块（myclient.c）

（1）TLS/SSL 初始化与证书验证模块

1）初始化 TLS 上下文，设置验证模式为 SSL_VERIFY_PEER。

2）加载 CA 证书目录，验证服务器证书的真实性。

3）实现 verify_callback() 回调函数，输出证书验证结果。

（2）TCP 客户端模块

1）根据传入的主机名和端口号建立 TCP 连接。

2）使用 gethostbyname() 解析服务器 IP（建议改用 getaddrinfo 以支持 IPv6）。

（3）TUN 设备与路由配置模块

1）创建 TUN 接口，并配置 IP 地址（如 192.168.53.X/24）。

2）添加路由规则，将目标为内网网段（192.168.60.0/24）的流量导向 TUN 接口。

（4）用户认证与数据收发模块

1）通过 SSL 发送用户名、密码及本地 TUN 接口 IP 尾号。

2）启动独立线程监听 TUN 接口，将本地流量通过 SSL 发送至服务器。

3）主线程接收服务器发来的加密数据，写入 TUN 接口。

2.1.3 工作机制

1. 隧道建立流程

（1）客户端解析服务器地址，建立 TCP 连接。

（2）双方进行 TLS 握手，客户端验证服务器证书。

（3）客户端发送用户名、密码和本地 TUN IP 尾号。

（4）服务器验证用户信息，若成功则为该客户端创建管道文件。

（5）隧道建立完成，双方开始监听 TUN 接口和 SSL 套接字。

2. 数据包流向（以客户端访问内网 HostV 为例）

（1）客户端应用程序（如 ping）发送数据包，目标 IP 为 192.168.60.101。

（2）系统根据路由表将数据包发往 TUN 接口。

（3）客户端 VPN 程序读取 TUN 数据，通过 SSL 加密后发送至服务器。

（4）服务器子进程接收加密数据，解密后写入 TUN 接口。

（5）服务器系统根据路由将数据包转发至内网（通过 docker2 接口）。

（6）返回流程类似，方向相反。

3. 多客户端支持机制

（1）服务器父进程监听 TUN，根据目标 IP 尾号选择对应客户端管道。

（2）每个客户端对应一个子进程和一个管道文件，实现流量隔离。

（3）使用 select() 或线程监听多个输入源（TUN、SSL、管道）。

2.2 详细设计

2.2.1 服务器设计

1.主要数据结构

（1）PIPEDATA 结构体

用于在线程间传递客户端专属的通信通道信息，每个成功登录的客户端会创建一个命名 FIFO 管道（如 ./pipe/192.168.53.105 中的 105），该结构将管道路径与 TLS 会话绑定，供 listen_pipe 线程使用。如图 2-3 所示。

```
79      typedef struct pipe_data {
80          char *pipe_file;
81          SSL *ssl;
82      } PIPEDATA;
```

图 2-3 PIPEDATA 数据结构

（2）其它结构体

代码使用了操作系统和标准库提供的预定义结构体：struct ifreq 用于 TUN 设备配置，struct sockaddr_in 用于 TCP 套接字地址，struct spwd 用于用户认证。这些结构体通过相应的头文件引入，代码只需声明其实例而不需要重新定义类型。

2.重要算法与机制

（1）服务器认证客户端

服务器利用系统 /etc/shadow 文件中存储的账户信息对客户端提交的登录凭据进行身份验证，具体流程如下：

1）获取用户加密密码：服务器首先调用 getspnam(user_name) 函数，根据客户端提供的用户名查询系统中的 shadow 条目。若该用户存在，函数将返回一个包含其加密密码（即散列值）的结构体；若用户不存在，则返回 NULL。

2）密码比对验证：服务器使用 crypt() 函数对客户端提交的明文密码进行加密。该函数会

自动从 /etc/shadow 中获取对应用户的盐值（salt），并采用相同的哈希算法（如 SHA-512）生成加密结果。随后，将此结果与 shadow 文件中存储的散列值进行严格比对。

3）认证结果处理：若用户名不存在，或加密后的密码与 shadow 中记录不一致，则判定为认证失败，服务器返回相应的错误信息（如“用户不存在”或“密码错误”）。若用户名有效且密码哈希完全匹配，则认证成功，服务器返回登录成功的提示，并允许客户端继续后续通信流程。

该机制依赖于 Linux 系统原生的身份验证体系，确保了只有拥有合法系统账户及正确密码的用户才能接入服务。代码如图 2-4 所示。

```
104     int login(char *user, char *passwd) {
105         struct spwd *pw;
106         char *epasswd;
107         pw = getspnam(user);
108         if (pw == NULL) {
109             printf("Error: Password is NULL.\n");
110             return 0;
111         }
112 
113         printf("USERNAME : %s\n", pw->sp_namp);
114         printf("PASSWORD : %s\n", pw->sp_pwdp);
115 
116         epasswd = crypt(passwd, pw->sp_pwdp);
117         if (strcmp(epasswd, pw->sp_pwdp)) {
118             printf("Error: The password is incorrect!\n");
119             return 0;
120         }
121         return 1;
```

图 2-4 服务器认证客户端

（2）多客户端服务

服务端通过命名管道（FIFO）实现与各客户端之间的下行数据通信，具体流程如下：

1）子进程与命名管道的创建：

当客户端成功建立连接后，服务端会 fork 一个子进程专门处理该客户端的会话。随后，根据分配给该客户端的虚拟 IP 地址（格式为 192.168.53.x），提取其最后一字节 x（例如 IP 为 192.168.53.105，则 x＝105），并在 ./pipe/ 目录下创建一个名为 x 的命名管道（如 ./pipe/105）。该管道将作为从服务端向该客户端转发网络数据的专用通道。

2）报文监听与定向写入：

服务端在主线程中已启动一个 listen_tun() 线程，持续从 TUN 虚拟网卡读取内核投递的 IP 数据包。对于每个捕获到的数据包，若其目标地址属于 192.168.53.0/24 子网，则解析出目标 IP 的最后一个字节，并据此确定对应的命名管道路径。随后，服务端调用 write() 函数，将整个 IP 数据包写入该命名管道。这样，每个客户端专属的 listen_pipe 线程便能从自己的 FIFO 中读取数据，并通过 TLS 安全地发送回客户端。此机制实现了从内核网络栈到特定客户端的精准下行数据分发。

```
85      void *listen_tun(void *tunfd) {
86          int fd = *((int *)tunfd);
87          char buff[2000];
88          while (1) {
89              int len = read(fd, buff, 2000);
90              if (len > 19 && buff[0] == 0x45) {
91                  printf("TUN Received, length : %d , destination : 192.168.53.%d\n", len, (int)
92                  char pipe_file[10];
93                  sprintf(pipe_file, "./pipe/%d", (int)buff[19]);
94                  int fd = open(pipe_file, O_WRONLY);
95                  if (fd == -1) {
96                      printf("[WARNING] File %s does not exist.\n", pipe_file);
97                  } else {
98                      write(fd, buff, len);
99                  }
100             }
101         }
```

图 2-5 多客户端服务

（3）发送数据

sendOut() 函数是实现客户端到外部网络通信的核心环节，主要负责将客户端通过 TLS 通道发送的数据注入服务器的内核网络栈，从而完成上行数据转发。其工作流程如下：

1)接收客户端数据：sendOut() 首先从已建立的 TLS 连接中读取客户端发来的原始数据。这些数据通常是完整的 IP 数据包，代表客户端希望经由服务端访问内部或外部网络的请求。

2）写入 TUN 虚拟设备：接收到的数据会被直接写入到之前创建的 TUN 虚拟网络接口（如 tun0）。TUN 设备在 Linux 中表现为一个字符设备，用户空间程序向其写入的数据会被内核视为“从该虚拟网卡接收到的 IP 包”。

3）内核路由与转发：一旦数据包进入 TUN 设备，Linux 内核便会按照标准网络协议栈对其进行处理：根据目标 IP 地址查询路由表，并将数据包转发至相应的目的地（如同一子网内的其他主机，或通过网关访问外网）。由此，客户端便能通过这条安全隧道透明地访问远程

网络资源，实现 VPN 的基本功能。

```
138     void sendOut(SSL *ssl, int sock, int tunfd) {
139         int len;
140         do {
141             char buf[1024];
142             len = SSL_read(ssl, buf, sizeof(buf) - 1);
143             write(tunfd, buf, len);
144             buf[len] = '\0';
145             printf("SSL Received : %d\n", len);
146         } while (len > 0);
147         printf(" SSL shutdown.\n");
148     }
```

图 2-6 发送数据

（4）监听管道

listen_pipe() 函数是服务端实现下行数据通信（即从服务器向客户端发送数据）的关键组件，其核心职责是从命名管道中读取待发往客户端的数据，并通过安全通道将其转发。具体功能如下：

1）监听专属命名管道：该函数接收一个 PIPEDATA 结构体作为参数，其中包含指向特定命名管道（如 ./pipe/105）的路径和对应客户端的 SSL 连接上下文。函数会打开该命名管道并持续监听，等待内核或 TUN 线程写入待转发的 IP 数据包。

2）读取待转发数据：一旦管道中有数据可读（通常是由 listen_tun() 线程根据目标 IP 写入的完整 IP 包），listen_pipe() 便立即读取这些数据。这些数据代表了应发往该客户端的网络流量，例如来自内网其他主机的响应或服务端生成的 ICMP 包等。

3）通过 TLS 安全回传：读取到的数据随后通过已建立的 SSL/TLS 连接，使用 SSL_write() 发送回对应的客户端。这一过程不仅保障了传输的机密性与完整性，也完成了从虚拟网络接口经由安全隧道返回客户端的完整下行通路。

```
124     void *listen_pipe(void *threadData) {
125         PIPEDATA *ptd = (PIPEDATA*)threadData;
126         int pipefd = open(ptd->pipe_file, O_RDONLY);
127 
128         char buff[2000];
129         int len;
130         do {
131             len = read(pipefd, buff, 2000);
132             SSL_write(ptd->ssl, buff, len);
133         } while (len > 0);
134         printf("%s read 0 byte. Connection closed and file removed.\n", ptd->pipe_file);
135         remove(ptd->pipe_file);
136     }
```

图 2-7 监听管道

3.主要函数流程图

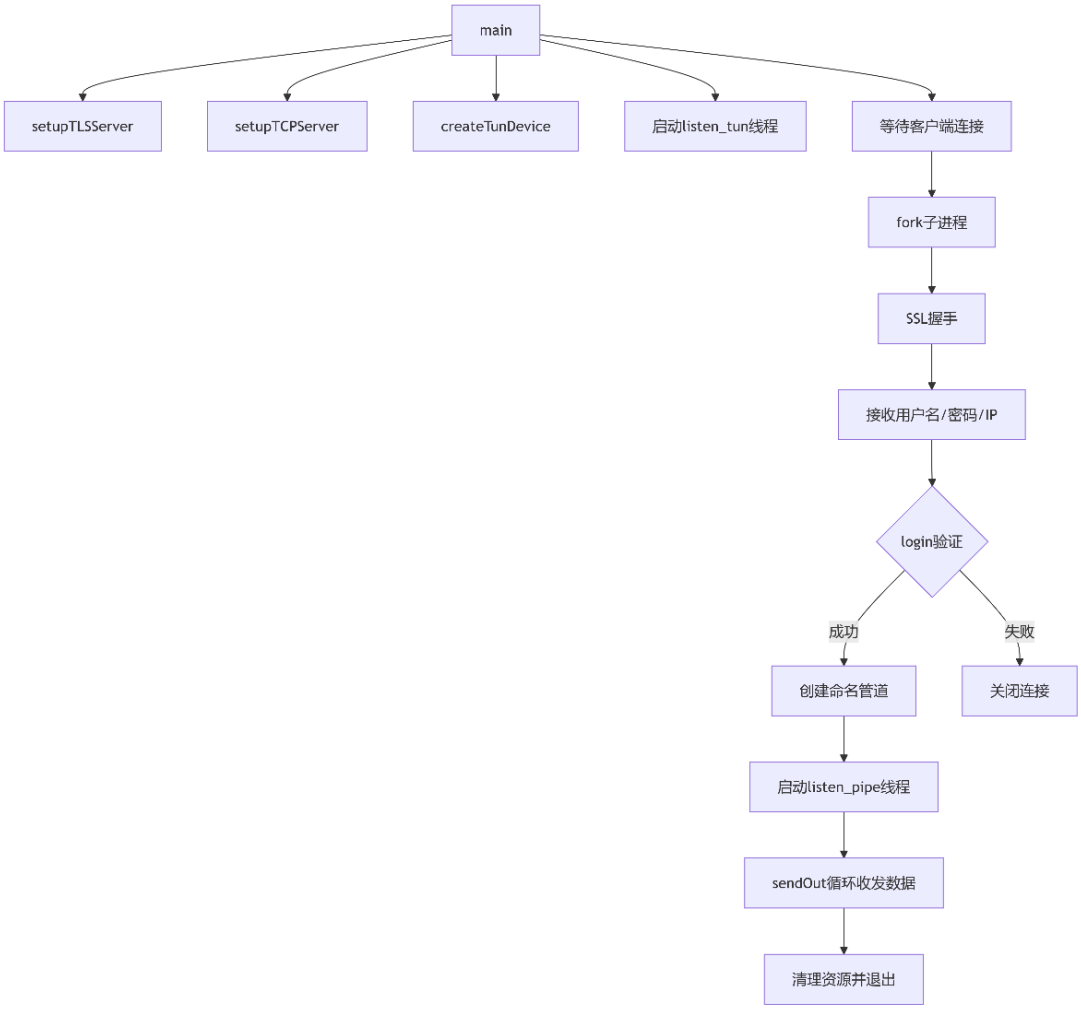

图 2-8 myserver.c 主要函数流程图

4.关键代码

（1）SSL/TLS 初始化

初始化 TLS 上下文，加载服务器证书和私钥，支持 TLSv1.2，如图 2-9 所示。

```
28  SSL *setupTLSServer() {
29      SSL_METHOD *meth;
30      SSL_CTX *ctx;
31      SSL *ssl;
32
33      SSL_library_init();
34      SSL_load_error_strings();
35      SSLeay_add_ssl_algorithms();
36      meth = (SSL_METHOD *)TLSv1_2_method();
37      ctx = SSL_CTX_new(meth);
38
39      SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);
40      ////////////////////////////////////////////////////////////////////change
41      SSL_CTX_use_certificate_file(ctx, "./cert/server-lxe.crt", SSL_FILETYPE_PEM);
42      SSL_CTX_use_PrivateKey_file(ctx, "./cert/server-lxe.key", SSL_FILETYPE_PEM);
43      ssl = SSL_new(ctx);
44
45      return ssl;
46  }
```

图 2-9 SSL/TLS 初始化

（2）TUN 设备创建与监听

监听 TUN 设备，解析 IP 包（IPv4 协议号 0x45），根据目标 IP 最后一字节转发到对应管道。如图 2-10 所示。

```
85  void *listen_tun(void *tunfd) {
86      int fd = *((int *)tunfd);
87      char buff[2000];
88      while (1) {
89          int len = read(fd, buff, 2000);
90          if (len > 19 && buff[0] == 0x45) {
91              printf("TUN Received, length : %d , destination : 192.168.53.%d\n", len, (int)buff[19]);
92              char pipe_file[10];
93              sprintf(pipe_file, "./pipe/%d", (int)buff[19]);
94              int fd = open(pipe_file, O_WRONLY);
95              if (fd == -1) {
96                  printf("[WARNING] File %s does not exist.\n", pipe_file);
97              } else {
98                  write(fd, buff, len);
99              }
100         }
101     }
102 }
```

图 2-10 TUN 设备创建与监听

（3）用户登录验证

使用系统影子密码进行用户认证，使用 crypt() 加密比对密码。如图 2-11 所示。

```
104     int login(char *user, char *passwd) {
105         struct spwd *pw;
106         char *epasswd;
107         pw = getspnam(user);
108         if (pw == NULL) {
109             printf("Error: Password is NULL.\n");
110             return 0;
111         }
112 
113         printf("USERNAME : %s\n", pw->sp_namp);
114         printf("PASSWORD : %s\n", pw->sp_pwdp);
115 
116         epasswd = crypt(passwd, pw->sp_pwdp);
117         if (strcmp(epasswd, pw->sp_pwdp)) {
118             printf("Error: The password is incorrect!\n");
119             return 0;
120         }
121         return 1;
122     }
```

图 2-11 用户登录验证

（4）管道监听与 SSL 发送

从命名管道读取数据，并通过 SSL 连接发送到客户端。如图 2-12 所示。

```
124     void *listen_pipe(void *threadData) {
125         PIPEDATA *ptd = (PIPEDATA*)threadData;
126         int pipefd = open(ptd->pipe_file, O_RDONLY);
127 
128         char buff[2000];
129         int len;
130         do {
131             len = read(pipefd, buff, 2000);
132             SSL_write(ptd->ssl, buff, len);
133         } while (len > 0);
134         printf("%s read 0 byte. Connection closed and file removed.\n", ptd->pipe_file);
135         remove(ptd->pipe_file);
136     }
```

图 2-12 管道监听与 SSL 发送

（5）主服务循环

主进程监听 TCP 连接，为每个客户端创建子进程处理，实现并发服务。如图 2-13 所示。

```
168         while (1) {
169             int sock = accept(listen_sock, (struct sockaddr *)&sa_client, &client_len);  // block, if try connect
170             if (fork() == 0) {                                                            // The child process
```

图 2-13 主服务循环

2.2.2 客户端设计

1.主要数据结构

THDATA 结构体

用于传递线程参数，包含：tunfd：TUN 设备的文件描述符，用于读写虚拟网络数据包。ssl：SSL/TLS 连接对象，用于安全通信。

```
100  typedef struct thread_data
101  {
102      int tunfd;
103      SSL *ssl;
104  } THDATA, *PTHDATA;
```

图 2-14 THDATA 结构体

2.重要算法与机制

（1）客户端认证服务器

客户端通过调用 SSL_CTX_load_verify_locations() 函数，从指定的 "ca_client" 目录中加载用于验证服务器身份的 CA 证书。该函数依据证书文件名的哈希值来查找并匹配相应的证书，从而为后续的验证提供可信的根证书数据。在 SSL 连接建立过程中，客户端利用已配置的 SSL 上下文执行完整的服务器证书验证流程，确保通信对端的身份真实可信，从而保障整个连接的安全性与可靠性。

```
44      SSL *setupTLSClient(const char *hostname)
45      {
46
47          SSL_library_init();
48          SSL_load_error_strings();
49          SSLeay_add_ssl_algorithms();
50          SSL_METHOD *meth;
51          SSL_CTX *ctx;
52          SSL *ssl;
53          meth = (SSL_METHOD *)TLSv1_2_method();
54          ctx = SSL_CTX_new(meth);
55
56          SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
57          if (SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1)
58          {
59              printf("Error: Error setting the verify locations. \n");
60              exit(0);
61          }
62          ssl = SSL_new(ctx);
63
64          X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
65          X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
66
67          return ssl;
68      }
```

图 2-15 setupTLSClient 函数

为进一步提升验证机制的灵活性和兼容性，客户端还注册了一个自定义的证书验证回调函数。该回调不仅用于展示证书的基本信息，还承担了更细致的验证逻辑。在处理验证结果时，该回调采用相对宽松的策略：对于某些非关键性问题（例如服务器主机名与证书中 Subject Alternative Name 或 Common Name 不完全匹配等轻微不一致），它会选择忽略这些错误，而非直接中断连接。这种设计在维持基本安全要求的前提下，增强了系统对不同部署环境和配置差异的适应能力。

通过上述机制，客户端既严格遵循了安全验证的核心原则，又兼顾了实际应用中的多样性和容错需求，从而提升了 VPN 连接的稳定性与用户体验。

```
23  int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
24  {
25      char buf[300];
26
27      X509 *cert = X509_STORE_CTX_get_current_cert(x509_ctx);
28      X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
29      printf("subject = %s\n", buf);
30
31      if (preverify_ok == 1)
32      {
33          printf("Verification passed.\n");
34      }
35      else
36      {
37          int err = X509_STORE_CTX_get_error(x509_ctx);
38          printf("Error: Verification failed: %s.\n", X509_verify_cert_error_string(err));
39      }
40
41      return preverify_ok;
42  }
```

图 2-16 verify_callback 函数

（2）TUN 创建

在 Linux 操作系统中，设备是作为一种特殊的文件类型进行操作的。因此可以通过创建设备目录下的设备来创建 Tun0 虚拟网卡。

```
86      int createTunDevice()
87      {
88          int tunfd;
89          struct ifreq ifr;
90          memset(&ifr, 0, sizeof(ifr));
91
92          ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
93
94          tunfd = open("/dev/net/tun", O_RDWR);
95          ioctl(tunfd, TUNSETIFF, &ifr);
96
97          return tunfd;
98      }
```

图 2-17 createTunDevice 函数

3.主要函数流程图

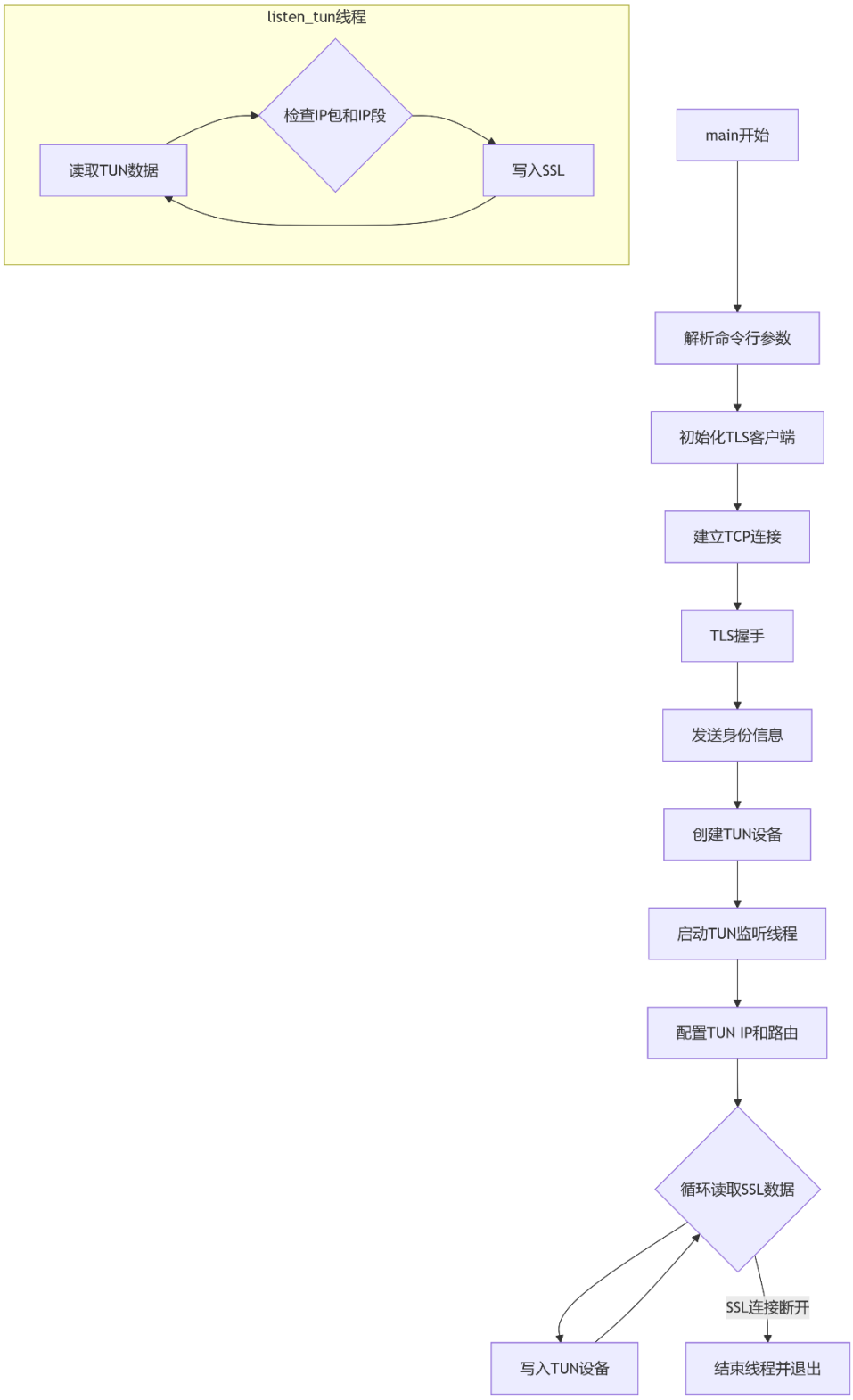

图 2-18 myclient.c 主函数流程图

4.关键代码

（1）TLS 客户端初始化

1）使用 TLS 1.2 协议。

2）设置证书验证回调函数和 CA 目录。

3）启用主机名验证，防止中间人攻击。

```
44      SSL *setupTLSClient(const char *hostname)
45      {
46
47          SSL_library_init();
48          SSL_load_error_strings();
49          SSLeay_add_ssl_algorithms();
50          SSL_METHOD *meth;
51          SSL_CTX *ctx;
52          SSL *ssl;
53          meth = (SSL_METHOD *)TLSv1_2_method();
54          ctx = SSL_CTX_new(meth);
55
56          SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
57          if (SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1)
58          {
59              printf("Error: Error setting the verify locations. \n");
60              exit(0);
61          }
62          ssl = SSL_new(ctx);
63
64          X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
65          X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
66
67          return ssl;
68      }
```

图 2-19 TLS 客户端初始化

（2）TUN 设备数据包过滤

1）buff[0] == 0x45 表示 IPv4 包。

2）buff[15] 是目标 IP 的最后一段，与 last_ip 比较。

3）只有匹配的 IP 包才会通过 SSL 发送。

```
118             if (len > 19 && buff[0] == 0x45)
119             {
120                 if ((int)buff[15] == atoi(last_ip))
121                 {
122                     printf("Receive TUN: %d\n", len);
123                     SSL_write(ptd->ssl, buff, len);
124                 }
```

图 2-20 TUN 设备数据包过滤

（3）身份认证与隧道建立

1）客户端在 TLS 连接建立后立即发送用户名、密码和 IP 段。

2）服务器端验证通过后，隧道正式建立。

```
165         SSL_write(ssl, argv[3], strlen(argv[3])); // username
166         SSL_write(ssl, argv[4], strlen(argv[4])); // password
167         SSL_write(ssl, argv[5], strlen(argv[5])); // local the last byte of IP
```

图 2-21 身份认证与隧道建立

（4）TUN 设备与路由配置

1）为 tun0 分配 IP 地址。

2）添加路由规则，将目标网段 192.168.60.0/24 路由到 tun0。

```
179         sprintf(cmd, "sudo ifconfig tun0 192.168.53.%s/24 up && sudo route add -net 192.168.60.0/24 tun0",
180                 argv[5]);
181         system(cmd);
```

图 2-22 TUN 设备与路由配置

3 VPN 实现细节

3.1 隧道断开对通信的影响分析

当服务器端终止其服务程序时，VPN 隧道连接会被断开，此时通过 Telnet 发送的命令将无法送达。然而，一旦服务器和客户端重新启动，Telnet 连接即可恢复，并且此前未能成功发送的命令会自动重发并被执行。

要确保 VPN 隧道能够顺利断开并重新建立，必须同时中断服务端和客户端的连接。这是因为在隧道建立过程中，服务端会启动并进入等待客户端连接请求的状态；一旦收到请求，便会初始化相应的服务进程，并持续运行于数据转发循环中。如果仅关闭客户端，服务端仍停留在该循环中，继续监听原连接端口，无法感知连接已失效，因而无法接受新的连接请求。反之，若只关闭服务端而客户端保持运行，客户端仍处于原有的隧道处理逻辑中，由于其本地发送端口未变，也无法主动发起新的连接尝试。

因此，只有在服务端与客户端同时断开的情况下，双方才能彻底退出当前状态，完成必要的清理和重置，从而为下一次连接做好准备。

在重新建立隧道时，客户端首先向服务端发起新的连接请求，并接收服务端返回的响应数据包。由于 Telnet 基于 TCP 协议，当连接中断时，未成功传输的数据会触发 TCP 的超时重传机制。一旦连接重建，服务端因仍处于等待确认的状态，会发送 ACK 以确认之前丢失的数据段。当客户端收到三个重复的 ACK（即冗余 ACK）时，会激活快速重传机制，立即重发那些未被确认的报文。服务端收到这些重传数据后进行处理，并返回响应；客户端则据此完成命令的回显。

这一系列 TCP 可靠传输机制确保了即使在连接中断后，先前输入的命令也能在恢复连接时被正确重传、执行和显示，从而保障了 Telnet 会话的数据一致性与完整性。

3.2 VPN 登录协议设计说明

在 SSL 连接成功建立之后、正式进入隧道数据传输之前，我们的 SSL VPN 系统会执行一个简明而可靠的登录协议，用于完成客户端与服务器之间的会话初始化。该协议完全运行在已加密的 SSL 通道之上，确保控制信令的安全性。

整个登录过程始于客户端向服务器发送一个名为 LOGIN_REQ 的登录请求报文。该报文

采用二进制格式，由 1 字节类型字段、2 字节长度字段以及可选的载荷组成。其中，类型字段固定为 0x01，长度字段指示后续载荷的字节数（当前实现中通常为 0，表示无附加信息），用于未来扩展支持版本协商或轻量级认证信息。此报文的作用是明确告知服务器：客户端已完成 SSL 握手，现请求进入 VPN 隧道模式。

服务器收到 LOGIN_REQ 后，会根据当前策略、资源状态及连接合法性进行判断。若一切正常，服务器将回复一个 LOGIN_ACK 报文。该报文同样采用二进制结构，类型字段为 0x02，长度为 1，其后紧跟 1 字节的状态码 0x00，表示“接受连接”。客户端收到此确认后，即启动隧道数据转发逻辑，开始收发封装后的网络流量。

若服务器因任何原因（如并发连接数超限、客户端 IP 被拒绝、或协议版本不匹配等）无法接受该请求，则会返回一个 LOGIN_ERR 报文。该错误报文的类型字段为 0x03，长度字段包含后续错误代码和可读错误信息的总字节数。其内容由 1 字节错误码（例如 0x01 表示“拒绝连接”，0x02 表示“版本不支持”等）和一段以 ASCII 编码的错误描述字符串组成，便于客户端日志记录或用户提示。

整个报文交互严格遵循“请求—响应”顺序：客户端先发 LOGIN_REQ，服务器随后返回 LOGIN_ACK 或 LOGIN_ERR，且仅允许一次交互。这种设计避免了状态机复杂化，同时保证了连接建立过程的确定性和安全性。所有报文均通过 SSL_write 和 SSL_read 在已验证的 SSL 会话中传输，既继承了 SSL/TLS 提供的机密性与完整性保护，又通过结构化控制协议实现了可靠、可扩展的 VPN 登录机制。

4 问题分析与解决

4.1 权限不足无法访问 shadow 文件问题

4.1.1 问题描述

服务器身份验证时返回 NULL。

4.1.2 原因分析

非 root 用户无法读取/etc/shadow 文件。

4.1.3 解决方法

使用 sudo 权限运行服务器程序

```
sudo ./myserver
```

4.2 多客户端 IP 冲突问题

4.2.1 问题描述

多个客户端使用相同 IP 段导致路由混乱。

4.2.2 原因分析

手动指定 IP 容易冲突。

4.2.3 解决方法

实现自动 IP 分配机制（如上述设计方案）。

4.3 管道文件残留问题

4.3.1 问题描述

客户端异常断开后管道文件未清理。

4.3.2 原因分析

程序未处理异常退出情况。

4.3.3 解决方案

```c
// 添加信号处理和清理机制
void cleanup_handler(int sig) {
    remove(pipe_file);
    exit(0);
}
signal(SIGINT, cleanup_handler);
signal(SIGTERM, cleanup_handler);
```

图 4-1 管道文件残留清理

4.4 路由配置错误问题

4.4.1 问题描述

无法 ping 通目标主机。

4.4.2 原因分析

路由表配置不正确。

4.4.3 解决方案

在 HostV 上添加路由

```
route add -net 192.168.53.0/24 gw 192.168.60.1
```

5 体会与建议

5.1 心得体会

通过本次实验，我对 VPN 的实际实现机制有了远超课本理论的深入理解。亲手用 C 语言编写一个简易的 miniVPN 系统，让我真切体会到隧道技术如何打通内网与外网之间的通信路径，也掌握了数据在传输过程中是如何通过加密和身份认证来保障安全的。这种从代码层面接触协议流程的经历，极大加深了我对 VPN 数据流转逻辑的认识。

在实现多客户端支持的过程中，我综合运用了 Linux 下的多线程编程以及 select I/O 多路复用机制，这不仅解决了并发连接的问题，也让我对操作系统中进程调度、文件描述符管理和网络编程模型有了更扎实的掌握。整个开发过程成为一次理论与实践紧密结合的操作系统知识巩固之旅。

实验指导材料内容详实，从使用 Docker 搭建隔离的网络测试环境、生成 CA 与终端证书，到集成 TUN/TAP 虚拟网卡、实现 shadow 口令认证、应用管道通信及 select 事件驱动模型等关键环节，都提供了清晰的步骤说明和可运行的示例代码，为初学者构建了一条循序渐进的学习路径。

尤其值得思考的是实验中提出的问题，例如关于 Telnet 连接中断后自动恢复的现象。借助指导书中提供的 TCP 状态图和重传机制分析，我深入理解了 TCP 协议如何通过超时重传与快速重传来保障连接的可靠性，从而对网络服务的底层行为有了更本质的认知。

此外，我完整实践了基于 OpenSSL 构建私有 CA 的全过程——包括生成根证书、签发客户端/服务器证书、配置验证路径等。这一流程将原本抽象的 PKI 概念转化为具体操作，使我对数字证书的信任链机制有了切身体会。同时，利用 Wireshark 抓包分析加密流量，也显著提升了我在网络诊断与协议分析方面的实操能力。

在编写多进程版 VPN 服务器时，我遇到了并发控制与进程间通信的挑战。虽然示例使用了匿名管道（pipe()），但我在理解其原理后，主动尝试改用 mkfifo() 创建命名管道（FIFO）来实现父子进程之间的状态同步。这一调整不仅让我掌握了一种新的 IPC（进程间通信）方式，也锻炼了我在资源管理与错误处理方面的编程思维。我逐渐意识到，一个稳定可靠的服务器程序，离不开严谨的资源释放策略和完善的异常应对机制。

当然，部分代码确实与示例较为相似，但这在系统级编程中是合理的——因为像 TUN 设

备初始化、shadow 认证调用等操作本身就有标准实现范式。我的学习重点并非追求代码独创性，而是通过“阅读—理解—修改—验证”的方式，真正掌握每一行代码背后的技术原理。

总的来说，这次实验极大地丰富了我对 VPN 技术栈的理解，从加密认证、虚拟网络设备到并发模型与系统调用，多个知识点在实践中融会贯通。它不仅提升了我的工程实现能力，也点燃了我进一步探索主流开源 VPN 软件的兴趣。

5.2 意见建议

可以把容易出错的地方总结一下。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1）请人代做或冒名顶替者；

（2）替人做且不听劝告者；

（3）实验报告内容抄袭或雷同者；

（4）实验报告内容与实际实验内容不一致者；

（5）实验代码抄袭者。

作者签名：