# HTTPS 服务器

Author: myzhibei myzhibei@qq.com
Date: 2022-11-06 21:15:42
Description: HTTPS Server



**2022 年 11 月**

## 一、 实验内容

实现：使用 C 语言实现最简单的 HTTP 服务器

1、同时支持 HTTP（80 端口）和 HTTPS（443 端口），使用两个线程分别监听各自端口。

2、只需支持 GET 方法，解析请求报文，返回相应应答及内容。

| 需支持的状态码 | 场景 |
| --- | --- |
| 200 OK | 对于 443 端口接收的请求，如果程序所在文件夹存在所请求的文件，返回该状态码，以及所请求的文件 |
| 301 Moved Permanently | 对于 80 端口接收的请求，返回该状态码，在应答中使用 Location 字段表达相应的 https URL |
| 206 Partial Content | 对于 443 端口接收的请求，如果所请求的为部分内容（请求中有 Range 字段），返回该状态码，以及相应的部分内容 |
| 404 Not Found | 对于 443 端口接收的请求，如果程序所在文件夹没有所请求的文件，返回该状态码 |

## 二、 实验流程

1、根据实验内容，实现 HTTP 服务器程序。

2、执行 sudo python topo.py 命令，生成包括两个端节点的网络拓扑。

3、在主机 h1 上运行 HTTP 服务器程序，同时监听 80 和 443 端口。

```
h1 # ./http-server
```

4、在主机 h2 上运行测试程序，验证程序正确性。

```
h2 # python3 test/test.py
```

如果没有出现 AssertionError 或其他错误，则说明程序实现正确。

5、实现 HTTP 服务器分发视频。

6、在主机 h1 上运行 http-server，所在目录下有一个小视频（30 秒左右）

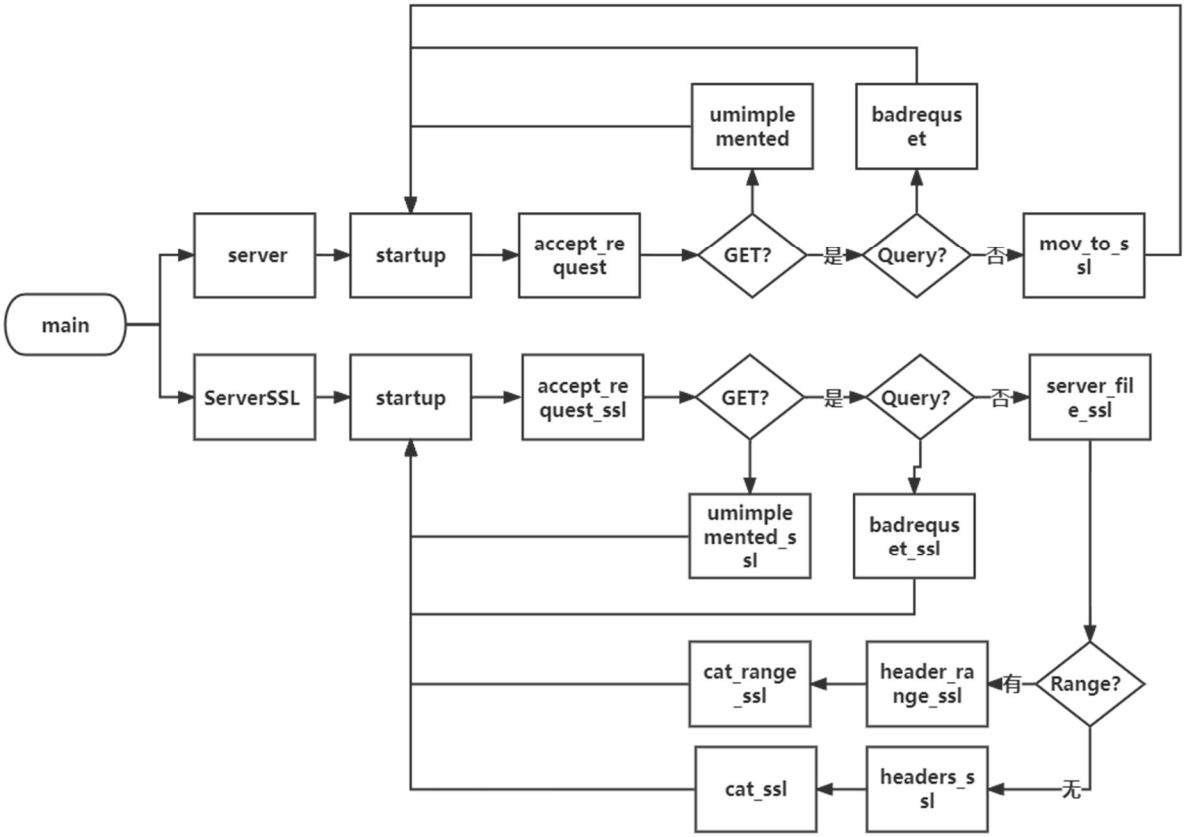
7、在主机 h2 上运行 vlc（注意切换成普通用户），通过网络获取并播放该小视频。

媒体 -> 打开网络串流 -> 网络 -> 请输入网络 URL -> 播放

8、通过抓包分析，说明 HTTP 服务器和 VLC 客户端是如何传输视频文件的。

## 三、 实验代码

1、代码总流程如图




2、所有函数声明如下：

```
const char *get_file_type(const char *);
void *accept_request(void *);
void *accept_request_ssl(void *);
void bad_request(SSL *);
void cat(int, FILE *);
void cat_ssl(SSL *, FILE *, int);
void cat_range_ssl(SSL *ssl, FILE *, int);
void error_die(const char *);
int get_line(int, char *, int);
void headers(int, const char *);
void headers_ssl(SSL *, const char *);
void not_found(int);
void not_found_ssl(SSL *);
```

```
    void serve_file(int, const char *);
    int startup(u_short *, int);
    void unimplemented(int);
    void unimplemented_ssl(SSL *);
    void mov_to_ssl(int, const char *);
    void serve_file_ssl(SSL *, char *, off_t , off_t , int );
    void do_error_ssl(SSL *, char *, char *, char *, char *);
```

3、详细代码分析

**（1） main**

使用 pthread 分别创建监听 80 端口的 Server 线程和 443 端口的 ServerSSL 两个线程，443 端口线程传递了使用的 SSL 证书和私钥，线程 pthread_join 阻塞主函数防止线程还没来得及执行主函数结束导致关闭。

```
    int main(void)
    {
        int port80 = 80;
        void *state;
        printf("Create 80 Server Thread...\n");
        pthread_t newServerThread;
        if (pthread_create(&newServerThread, NULL, Server,
&port80))
            perror("pthread_create 80");
        struct SSLargs sslarg;
        sslarg.port = 443;
        sslarg.lisnum = 2;
        strcpy(sslarg.ctf, "keys/cnlab.cert");
        strcpy(sslarg.prikey, "keys/cnlab.prikey");
        printf("Create 443 Server Thread...\n");
        if (pthread_create(&newServerThread, NULL, ServerSSL,
&sslarg))
            perror("pthread_create 443");

        pthread_join(newServerThread, &state);
        return (0);
    }
```

**（2） server**

根据创建线程传递的端口，执行 startup 函数创建 server socket，然后 while 循环接收客户端 socket 连接，接收到连接后创建一个新的 accept_request 处理函数线程处理。

```
    void *Server(void *arg)
    {
```

```
        u_short port = *(u_short *)arg;
        printf("MYZHIBEI Server Creating on port %d !\n", port);
        struct accSktArgs accSktarg;
        int server_sock = -1;
        int client_sock = -1;
        struct sockaddr_in client_name;
        socklen_t client_name_len = sizeof(client_name);
        pthread_t newAcceptThread;
        // 启动server socket

        server_sock = startup(&port, 5);
        accSktarg.server_sock = server_sock;

        printf("MYZHIBEI Server running on port %d !\n", port);

        while (1) {
            // 接受客户端连接
            accSktarg.client_sock = accept(server_sock, (struct
sockaddr *)&client_name, &client_name_len);
            if (accSktarg.client_sock == -1)
                error_die("accept");
            /*启动线程处理新的连接 */
            printf("HTTP Server: got connection from %s, port %d,
socket %d\n",
                   inet_ntoa(client_name.sin_addr),
                   ntohs(client_name.sin_port), server_sock);
            if (pthread_create(&newAcceptThread, NULL,
accept_request, &accSktarg) != 0)
                perror("pthread_create");
        }
        // 关闭server socket
        close(server_sock);
        printf("MYZHIBEI Server closed on port %d !\n", port);
        return NULL;
    }
```

### （3） ServerSSL

首先载入 SSL 库，创建 ctx，载入用户数字证书和私钥，检测是否有效后使用 startup 函数创建服务端 server socket，然后 while 循环接收 accept 客户端，接收到后创建 accept_request_ssl 函数线程处理消息。

```
    void *ServerSSL(void *args)
    {
```

```
        struct SSLargs sslarg = *(struct SSLargs *)args;
        u_short port = sslarg.port;
        int lisnum = sslarg.lisnum;
        SSL_CTX *ctx;
        SSL_library_init();
        OpenSSL_add_all_algorithms();   //载入所有 SSL 算法
        SSL_load_error_strings();     //载入所有 SSL 错误消息
        // 以 SSL V2 和 V3 标准兼容方式产生一个 SSL_CTX ，即 SSL
Content Text
        ctx = SSL_CTX_new(SSLv23_server_method());
        if (ctx == NULL) {
            error_die("ctx");
            exit(1);
        }
        if (SSL_CTX_use_certificate_file(ctx, sslarg.ctf,
SSL_FILETYPE_PEM) <= 0)
            error_die("ctx_certificate");
        if (SSL_CTX_use_PrivateKey_file(ctx, sslarg.prikey,
SSL_FILETYPE_PEM) <= 0)
            error_die("ctx_privateKey");
        if (!SSL_CTX_check_private_key(ctx))
            error_die("CTX_check_private_key");
        printf("MYZHIBEI https Server Creating on port %d !\n",
port);
        int server_sock = -1;
        int client_sock = -1;
        struct sockaddr_in client_name;
        socklen_t client_name_len = sizeof(client_name);
        pthread_t newAcceptSSLThread;
        // 启动server socket
        server_sock = startup(&port, lisnum);
        while (1) {
            struct sslAccArgs accarg;
            accarg.ctx = ctx;
            strcpy(accarg.rootdir, "");
            /* 等待客户端连上来 */
            client_sock = accept(server_sock, (struct sockaddr
*)&client_name, &client_name_len);
            if (client_sock == -1)
                error_die("accept");
            printf("HTTPS Server: got connection from %s, port %d,
socket %d to clientskt %d\n ",
```

```
                inet_ntoa(client_name.sin_addr),
                ntohs(client_name.sin_port), server_sock,
client_sock);
            accarg.client_socket = client_sock;
            if (pthread_create(&newAcceptSSLThread, NULL,
accept_request_ssl, &accarg))
                perror("pthread_create");
        }
        close(server_sock);
          SSL_CTX_free(ctx);
        return 0;
    }
```

**（4） startup**

根据本机地址创建 socket，绑定 port，若 port 不可用则随机分配一个端口，然后进行 listen 监听，返回 socket。

```
    int startup(u_short *port, int lisnum)
    {
        int httpskt = 0;
        // name,Structure describing an Internet socket address
        struct sockaddr_in name;
        // 设置http socket
        //  2,1,0 Create a new socket of type TYPE1 in domain
DOMAIN2, using protocol PROTOCOL0
        httpskt = socket(PF_INET, SOCK_STREAM, 0);
        if (httpskt == -1) //
            error_die("socket");
        memset(&name, 0, sizeof(name));
        name.sin_family = AF_INET;
        name.sin_port = htons(*port);

        name.sin_addr.s_addr = htonl(INADDR_ANY);

        printf("binding port:%d addr:%d\n", *port, INADDR_ANY);
        // 绑定端口
        if (bind(httpskt, (struct sockaddr *)&name, sizeof(name))
< 0)
            error_die("bind");
        if (*port == 0) /*动态分配一个端口 */
        {
            socklen_t namelen = sizeof(name);
```

```
            if (getsockname(httpskt, (struct sockaddr *)&name,
&namelen) == -1)
                error_die("getsockname");
            *port = ntohs(name.sin_port);
        }
        // 监听连接
        if (listen(httpskt, lisnum) < 0)
            error_die("listen");
        return (httpskt);
    }
```

### （5） accept_request

调用 get_line 函数获取 HTTP 报文的第一行数据，然后提取方法字段，若为不是 GET 方法则调用 umimplemented 函数告知客户端 501 状态未实现该方法，若是 GET 方法，获取 URL，再获取是否带有？查询参数，若带有，则调用 badrequset 函数告知 400 状态，若不带有，则调用 mov_to_ssl 函数告知客户端 301 状态重定向到 https 的地址，再将请求全部接受并丢弃，关闭该客户端连接。

```
    void *accept_request(void *accarg)
    {
        struct accSktArgs accArg = *(struct accSktArgs *)accarg;
        int client = accArg.client_sock;

        char buf[MAXBUF];
        int numchars;
        char method[255];
        char url[255];
        char path[512];
        size_t i, j;
        struct stat st;
        int cgi = 0; /* becomes true if server decides this is a
CGI program */
        char *query_string = NULL;
        // 获取一行HTTP 报文数据
        numchars = get_line(client, buf, sizeof(buf));
        printf("\nREQUEST === %s", buf);
        //
        i = 0;
```

```
        j = 0;
        // 对于HTTP 报文来说，第一行的内容即为报文的起始行，格式为
<method> <request-URL> <version>，
        // 每个字段用空白字符相连
        while (!ISspace(buf[j]) && (i < sizeof(method) - 1)) {
            // 提取其中的请求方式是GET 还是POST
            method[i] = buf[j];
            i++;
            j++;
        }
        method[i] = '\0';
        // 函数说明：strcasecmp()用来比较参数s1 和s2 字符串，比较时会
自动忽略大小写的差异。
        // 返回值：若参数s1 和s2 字符串相同则返回0。s1 长度大于s2 长
度则返回大于0 的值，s1 长度若小于s2 长度则返回小于0 的值。
        if (strcasecmp(method, "GET")) {
            // 仅实现了GET
            unimplemented(client);
            return NULL;
        }
        i = 0;
        // 将method 后面的后边的空白字符略过
        while (ISspace(buf[j]) && (j < sizeof(buf)))
            j++;
        // 继续读取request-URL
        while (!ISspace(buf[j]) && (i < sizeof(url) - 1) && (j <
sizeof(buf))) {
            url[i] = buf[j];
            i++;
            j++;
        }
        url[i] = '\0';
        // 如果是GET 请求，url 可能会带有?,有查询参数
        if (strcasecmp(method, "GET") == 0) {
            query_string = url;
            while ((*query_string != '?') && (*query_string !=
'\0'))
                query_string++;
            if (*query_string == '?') {
                // 将解析参数截取下来
                *query_string = '\0';
                query_string++;
```

```
            }
        }
        printf("GET URL=== %s\n", url);
        //=============================================
        // 开启http 转https
        char toURL[MAXBUF];
        sprintf(toURL, "https://%s%s", IPSTR, url);
        mov_to_ssl(client, toURL);
        //=============================================
        while ((numchars > 0) && strcmp("\n", buf)) // 将HTTP 请求
头读取并丢弃
            numchars = get_line(client, buf, sizeof(buf));
        close(client);
        printf(" Finished... \n");
        return NULL;
    }
```

**（6） getline**

获取 socket 的一行数据，只要发现 c 为\n,就认为是一行结束，如果读到\r,再用 MSG_PEEK 的方式读入一个字符，如果是\n，从 socket 读出，如果是下个字符则不处理，将 c 置为\n，结束。如果读到的数据为 0 中断，或者小于 0，也视为结束，c 置为\n。

```
int get_line(int sock, char *buf, int size)
{   int i = 0;
    char c = '\0';
    int n;

    while ((i < size - 1) && (c != '\n')) {
        n = recv(sock, &c, 1, 0);
        if (n > 0) {
            if (c == '\r') {
                n = recv(sock, &c, 1, MSG_PEEK); // 偷窥一个字节，如
果是\n 就读走
                if ((n > 0) && (c == '\n'))
                    recv(sock, &c, 1, 0);
                else
```

```
                    // 不是\n（读到下一行的字符）或者没读到，置 c 为\n
跳出循环,完成一行读取
                    c = '\n';
            }
            buf[i] = c;
            i++;
        } else
            c = '\n';
    }
    buf[i] = '\0';
    return (i);
    }
```

### （7） mov_to_ssl

发送给客户端 301 状态以及将转向的 https 地址放在 Location 字段中。

```
void mov_to_ssl(int client, const char *toURL)
{
    printf("301 Moved to %s!\n", toURL);
    char buf[MAXBUF];
    // 返回 301
    strcpy(buf, "HTTP/1.1 301 Moved Permanently\r\n");
    send(client, buf, strlen(buf), 0);
    strcpy(buf, SERVER_STRING);
    send(client, buf, strlen(buf), 0);
    sprintf(buf, "Content-Type: text/html\r\n");
    send(client, buf, strlen(buf), 0);
    sprintf(buf, "Location: %s\r\n", toURL);
    send(client, buf, strlen(buf), 0);
    strcpy(buf, "\r\n");
    send(client, buf, strlen(buf), 0);
    }
```

### （8） accept_request_ssl

根据 ctx 创建 SSL，并将连接用户的 socket 加入到 SSL，SSL_accept 建立 ssl 连接，SSL_read 接收客户端发来的消息，检测请求方法是否为 GET，若不是则调用 unimplemented_ssl 告知客户端 501 状态不支持该方

法，若是，则获取请求的 URL，然后解析是否有？查询参数，若有则调用 bad_request_ssl 告知 400 状态不支持，若无，查看 URL 是文件还是目录，如果是目录，默认设置为 index.html，然后检测文件是否存在，如果不存在调用 not_found_ssl 告知客户端 404 状态未找到该文件，若存在，检测对该文件是否有执行权限，若有执行权限，则调用 bad_request_ssl 告知客户端 400 状态不支持执行文件，若无，也即为静态文件，则检测请求是否有 Range 字段，若有则提取 Range 字段所代表的文件 offset 和 size，调用 server_file_ssl 发送文件并将是否有 range 字段置 1，若无，则直接调用 server_file_ssl 发送文件，最后关闭并释放 SSL 连接，关闭客户端 socket。

```
void *accept_request_ssl(void *accarg)
{
    struct sslAccArgs accArg = *(struct sslAccArgs *)accarg;
    int client = accArg.client_socket;
    SSL *ssl;
    ssl = SSL_new(accArg.ctx); /* 基于 ctx 产生一个新的 SSL */
    SSL_set_fd(ssl, client);   /* 将连接用户的 socket 加入到 SSL */
    printf("SSL accept %d return :%d \n", client,
SSL_accept(ssl));
    char buf[MAXBUF];
    int numchars;
    char method[255];
    char url[255];
    char path[512];
    size_t i, j;
    struct stat st;
    char *query_string = NULL;
    char sslrow[1024];
    numchars = SSL_read(ssl, buf, MAXBUF);
    if (numchars != 0) {
        printf("接收消息成功:\n'%s', 共%d 个字节的数据\n", buf,
numchars);
    }
    strcpy(sslrow, buf);
    printf("\n REQUEST === %s", sslrow);
    i = 0;
    j = 0;
```

```
    // 对于HTTP 报文来说，第一行的内容即为报文的起始行，格式为<method>
<request-URL> <version>，
    // 每个字段用空白字符相连
    while (!ISspace(sslrow[j]) && (i < sizeof(method) - 1)) {
        // 提取其中的请求方式是GET 还是POST
        method[i] = sslrow[j];
        i++;
        j++;
    }
    method[i] = '\0';
    printf("method === %s\n", method);
    // 函数说明：strcasecmp()用来比较参数s1 和s2 字符串，比较时会自动
忽略大小写的差异。
    // 返回值：若参数s1 和s2 字符串相同则返回0。s1 长度大于s2 长度则
返回大于0 的值，s1 长度若小于s2 长度则返回小于0 的值。
    if (strcasecmp(method, "GET")) {
        // 仅实现了GET 和POST
        unimplemented_ssl(ssl);
        printf("Finished...\n\n");
        SSL_shutdown(ssl); // 关闭 SSL 连接
        SSL_free(ssl);     // 释放 SSL */
        close(client);     // 因为http 是面向无连接的，所以要关闭
        return NULL;
    }
    i = 0;
    // 将method 后面的后边的空白字符略过
    while (ISspace(sslrow[j]) && (j < sizeof(sslrow)))
        j++;
    while (!ISspace(sslrow[j]) && (i < sizeof(url) - 1) && (j <
sizeof(sslrow))) {
        url[i] = sslrow[j];
        // printf("%ld %c , %ld %c \n",j,sslrow[j],i,url[i]);
        i++;
        j++;
    }
    url[i] = '\0';
    printf("URI=== %s\n", url);
    // 如果是GET 请求，url 可能会带有?,有查询参数
    if (strcasecmp(method, "GET") == 0) {
        query_string = url;
        while ((*query_string != '?') && (*query_string != '\0'))
            query_string++;
```

```
        if (*query_string == '?') {
            // 将解析参数截取下来
            *query_string = '\0';
            query_string++;
            printf("Query_string=== %s\n", query_string);
        }
    }
    // 以上已经将起始行解析完毕
    //  url 中的路径格式化到path
    printf("GET URL=== %s\n", url);
    sprintf(path, ".%s", url);
    // 如果path 只是一个目录，默认设置为首页index.html
    if (path[strlen(path) - 1] == '/')
        strcat(path, "index.html");
    printf("GET PATH === %s\n", path);
    // 函数定义: int stat(const char *file_name, struct stat
*buf);
    // 函数说明: 通过文件名filename 获取文件信息，并保存在buf 所指的结
构体stat 中
    // 返回值: 执行成功则返回0，失败返回-1，错误代码存于errno（需要
include <errno.h>）
    if (stat(path, &st) == -1) {
        //访问的网页不存在
        not_found_ssl(ssl);
        printf("Finished...\n\n");
        SSL_shutdown(ssl); // 关闭 SSL 连接
        SSL_free(ssl);     // 释放 SSL */
        close(client);
        return NULL;
    } else {
        printf("%s exist ! \n", path);
        // 如果访问的网页存在则进行处理
        if ((st.st_mode & S_IFMT) == S_IFDIR) // S_IFDIR 代表目录
            // 如果路径是个目录，那就将主页进行显示
            strcat(path, "/index.html");
        if ((st.st_mode & S_IXGRP) || (st.st_mode & S_IXOTH)) {
            // S_IXGRP:用户组具可执行权限
            // S_IXOTH:其他用户具可读取权限
            printf("400 bad request === %s \n", query_string);
            bad_request_ssl(ssl);
            printf("Finished...\n\n");
            SSL_shutdown(ssl); // 关闭 SSL 连接
```

```
            SSL_free(ssl);      // 释放 SSL */
            close(client);      // 因为http 是面向无连接的，所以要关闭
            return NULL;
        } else {
            off_t offset = 0;
            off_t size = -1;
            char strbuf[255];
            char res[3][255];
            long int intRes[2];
            intRes[0] = intRes[1] = 0L;
            int y = sscanf(sslrow, "%*[^R]%[^:]", res[0]);
            printf("Y:%d cmp:%d\n", &y, strcmp(res[0], "Range") ==
0);
            if (!(strcmp(res[0], "Range") == 0)) {
                printf("no range  : %s\n", res[0]);
                printf("%d\n", strcmp(res[0], "Range") == 0);
                serve_file_ssl(ssl, path, 0, 0, 0);
            } else {
                printf("range: %s\n", res[0]);
                int cnt = sscanf(sslrow, "%*[^=]%s", res[0]);
                printf("%s\n %ld-%ld\n", res[0], intRes[0],
intRes[1]);
                char tmp[255];
                strcpy(tmp, res[0]);
                printf("tmp  : %s\n", tmp);
                int cd = sscanf(tmp, "=%[0-9]-%[0-9]", res[1],
res[2]);
                // int cd = sscanf(tmp, "=%ld-%ld%s", intRes[0],
intRes[1], res[0]);
                printf("!!!=%s-%s\n", res[1], res[2]);
                offset = atoi(&res[1]);
                size = atoi(&res[2]) - offset;
                printf("%d  %d\n", offset, size);
                serve_file_ssl(ssl, path, offset, size, 1);
                // 将静态文件返回
                printf("serve file === %s \n", path);
                // serve_file_ssl(ssl, path, 0, -1);
            }
        }
    }
    printf("Finished...\n\n");
    SSL_shutdown(ssl); // 关闭 SSL 连接
```

```
    SSL_free(ssl);     // 释放 SSL */
    close(client);     // 因为http 是面向无连接的，所以要关闭
    return NULL;
    }
```

### （9） unimplemented_ssl

告知客户端 501 状态，请求的方法不支持。

```
void unimplemented_ssl(SSL *ssl)
{
    printf("501 Method Not Implemented");
    char buf[MAXBUF];
    // 发送 501 说明相应方法没有实现
    strcpy(buf, "%sHTTP/1.1 501 Method Not Implemented\r\n");
    sprintf(buf, "%s%s",buf,SERVER_STRING);
    sprintf(buf, "%sContent-Type: text/html\r\n", buf);
    sprintf(buf, "%s\r\n", buf);
    sprintf(buf, "%s<HTML><HEAD><TITLE>Method Not
Implemented\r\n", buf);
    sprintf(buf, "%s</TITLE></HEAD>\r\n", buf);
    sprintf(buf, "%s<BODY><P>HTTP request method not
supported.\r\n", buf);
    sprintf(buf, "%s</BODY></HTML>\r\n", buf);
    SSL_write(ssl, buf, strlen(buf));
    }
```

### （10） not_found_ssl

告知客户端 404 状态，请求的文件未找到。

```
void not_found_ssl(SSL *ssl)
{
    printf("404 Not Find!\n");
    char buf[MAXBUF];
    // 返回 404
    strcpy(buf, "HTTP/1.1 404 NOT FOUND\r\n");
    sprintf(buf, "%s%s", buf, SERVER_STRING);
    sprintf(buf, "%sContent-Type: text/html\r\n", buf);
    sprintf(buf, "%s\r\n", buf);
    sprintf(buf, "%s<HTML><TITLE>Not Found</TITLE>\r\n", buf);
    sprintf(buf, "%s<BODY><P>The server could not fulfill\r\n",
buf);
    sprintf(buf, "%syour request because the resource
specified\r\n", buf);
```

```
    sprintf(buf, "%sis unavailable or nonexistent.\r\n", buf);
    sprintf(buf, "%s</BODY></HTML>\r\n", buf);
    SSL_write(ssl, buf, strlen(buf));
}
```

### （11） bad_request_ssl

告知客户端 400 状态，该请求不支持。

```
void bad_request_ssl(SSL *ssl)
{
    char buf[MAXBUF];
    // 发送400
    strcpy(buf, "HTTP/1.1 400 BAD REQUEST\r\n");
    sprintf(buf, "%sContent-type: text/html\r\n", buf);
    sprintf(buf, "%s\r\n", buf);
    sprintf(buf, "%s<P>Your browser sent a bad request \r\n",
buf);
    SSL_write(ssl, buf, sizeof(buf));
    }
```

### （12） serve_file_ssl

给客户端发送静态文件。首先打开文件，如果文件打开失败，调用 404 返回，若打开成功，获取文件的大小，如果不是含有 Range 字段的请求，则直接调用 headers_ssl 告知客户端 200 状态并调用 cat_ssl 发送文件，若含有 Range 字段，则先判断是请求自 offset 之后的全部文件内容还是部分文件内容，根据文件大小设置 size 为要发送的内容大小，调用 header_range_ssl 函数发送 206 状态以及发送的大小，然后发送文件部分内容。

```
void serve_file_ssl(SSL *ssl, char *fin_path, off_t offset, off_t
size, int range)
{
    printf("Sending File=== %s\n", fin_path);
    FILE *resource = NULL;
    // 打开文件
    resource = fopen(fin_path, "rb");
    if (resource == NULL)
        // 如果文件不存在，则返回not_found
        do_error_ssl(ssl, fin_path, "404", "Not Found", "Server
can't find the file");
```

```
    else {
        // offset TO all
        struct stat sizefile;
        int sizeoffile;
        if (stat(fin_path, &sizefile) == 0) {
            printf("file1 size = %d\n", sizefile.st_size);
            sizeoffile = sizefile.st_size;
        }
        if (range) {
            char buf[MAXBUF]; //= {0};
            if (size < 1) {

                size = sizeoffile - offset;
            }
            headers_range_ssl(ssl, fin_path, offset, size);
            printf("sending file offset=%d size=%d\n", offset,
size);
            // 读取文件到buf 中
            fseek(resource, offset, SEEK_SET);                // 光
标移到文件开始起第offset 个字节处。
            while (!feof(resource) && size > sizeof(buf) - 1) //
判断文件是否读取到末尾
            {
                // 读取并发送文件内容
                fread(buf, sizeof(char), sizeof(buf) - 1,
resource);
                printf("sended %ld / %ld batyes : \n", sizeof(buf)
- 1, size);
                SSL_write(ssl, buf, sizeof(buf) - 1);
                size = size - (sizeof(buf) - 1);
            }
            if ((size < sizeof(buf) - 1) && size > 0) {
                fread(buf, sizeof(char), size, resource);
                printf("sended %ld / %ld batyes : \n", size,
size);
                SSL_write(ssl, buf, size);
            }
        } else {
            // 添加HTTP 头
            headers_ssl(ssl, fin_path);
            // 并发送文件内容
            cat_ssl(ssl, resource, sizeoffile);
```

```
        }
    }
    fclose(resource); // 关闭文件句柄
    }
```

### （13） headers_ssl

发送响应头，状态为 200，以及调用 get_file_type 获取文件类型放入 Content-type 字段中。

```
void headers_ssl(SSL *ssl, const char *filename)
{
    char buf[1024];
    const char *dot_pos = strchr(filename + 1, '.');
    const char *file_type = get_file_type(dot_pos);
    printf("file %s type %s=== %s\n", filename, dot_pos,
file_type);
    strcpy(buf, "HTTP/1.1 200 OK\r\n");
    sprintf(buf, "%s%s", buf, SERVER_STRING);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, file_type);
    SSL_write(ssl, buf, strlen(buf));
    }
```

### （14） get_file_type

通过对文件后缀的解析，枚举获取文件类型。

```
typedef struct
{
    const char *type;
    const char *value;
} mime_type_t;
mime_type_t mime[] =
    {
        {".html", "text/html"},
        {".xml", "text/xml"},
        {".xhtml", "application/xhtml+xml"},
        {".txt", "text/plain"},
        {".rtf", "application/rtf"},
        {".pdf", "application/pdf"},
        {".word", "application/msword"},
        {".png", "image/png"},
        {".gif", "image/gif"},
        {".jpg", "image/jpeg"},
```

```
        {".jpeg", "image/jpeg"},
        {".au", "audio/basic"},
        {".mpeg", "video/mpeg"},
        {".mpg", "video/mpeg"},
        {".mp4", "video/mp4"},
        {".avi", "video/x-msvideo"},
        {".gz", "application/x-gzip"},
        {".tar", "application/x-tar"},
        {".css", "text/css"},
        {NULL, "text/plain"}};
const char *get_file_type(const char *type)
{
    if (type == NULL)
        return "text/plain";
    int i;
    for (i = 0; mime[i].type != NULL; ++i) {
        if (strcmp(type, mime[i].type) == 0)
            return mime[i].value;
    }
    return mime[i].value;
    }
```

**（15） cat_ssl**

直接通过 SSL_write 发送文件全部内容。

```
void cat_ssl(SSL *ssl, FILE *resource, int sizeoffile)
{
    char buf[MAXBUF] = {0};
    char str[1024];
    while (!feof(resource)) // 判断文件是否读取到末尾
    {
        // 读取并发送文件内容
        fread(buf, sizeof(char), sizeof(buf) - 1, resource);
        strcpy(str, buf);
        printf("sended %ld / %ld batyes : \n", sizeof(buf) - 1,
sizeoffile);
        SSL_write(ssl, str, sizeof(str) - 1);
    }
    }
```

**（14）headers_range_ssl**

发送响应头，状态码为 206，Content-type 为通过 get_file_type 获取的文件类型，Content-Length 字段为发送的部分文件的长度，Content-Range 字段为发送的部分文件起始和终点位置以及全长。

```
void headers_range_ssl(SSL *ssl, const char *filename, off_t
offset, off_t size)
{
    char buf[MAXBUF];
    const char *dot_pos = strchr(filename + 1, '.');
    const char *file_type = get_file_type(dot_pos);
    printf("file type === %s\n", file_type);
    strcpy(buf, "HTTP/1.1 206 Partial Content\r\n");
    sprintf(buf, "%s%s", buf, SERVER_STRING);
    sprintf(buf, "%sContent-type: %s\r\n", buf, file_type);
    sprintf(buf, "%sContent-Length: %d\r\n", buf, size);
    sprintf(buf, "%sContent-Range: bytes %d-%d/%d\r\n", buf,
offset, offset + size, size);
    sprintf(buf, "%s\r\n", buf);
    printf("range header === %s\n", buf);
    SSL_write(ssl, buf, strlen(buf));
    }
```

## 四、 实验环境

Ubuntu 20.04.5 LTS 64bit，OpenSSL 1.1.1f，Mininet 2.3.0.dev6

## 五、 实验过程

1、编译代码

gcc -W -Wall -o httpsServer myzhibei_https_server.c -lpthread -lssl -lcrypto

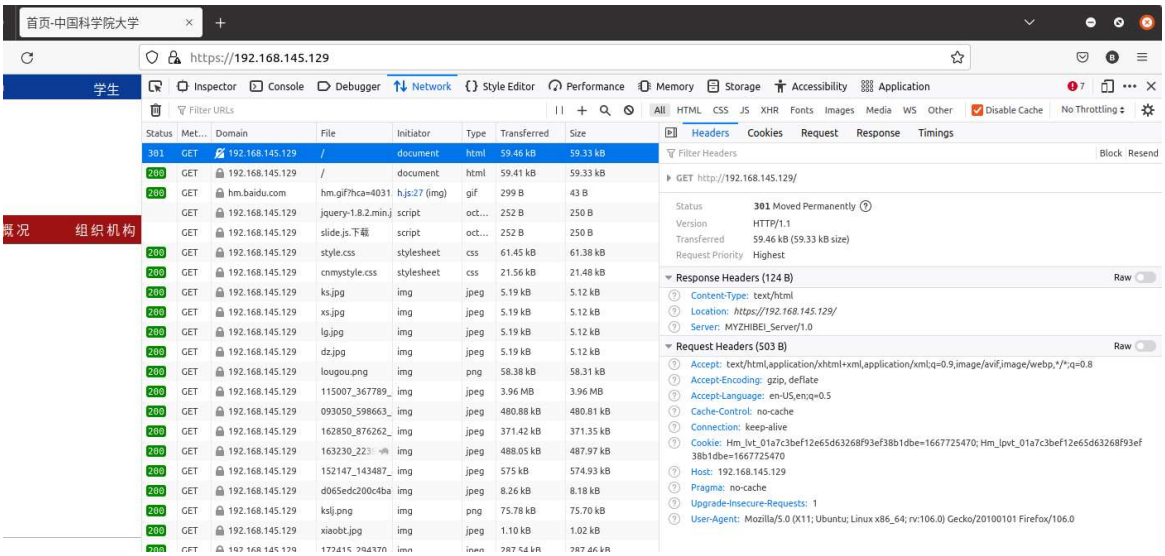
2、执行代码

sudo ./httpsServer

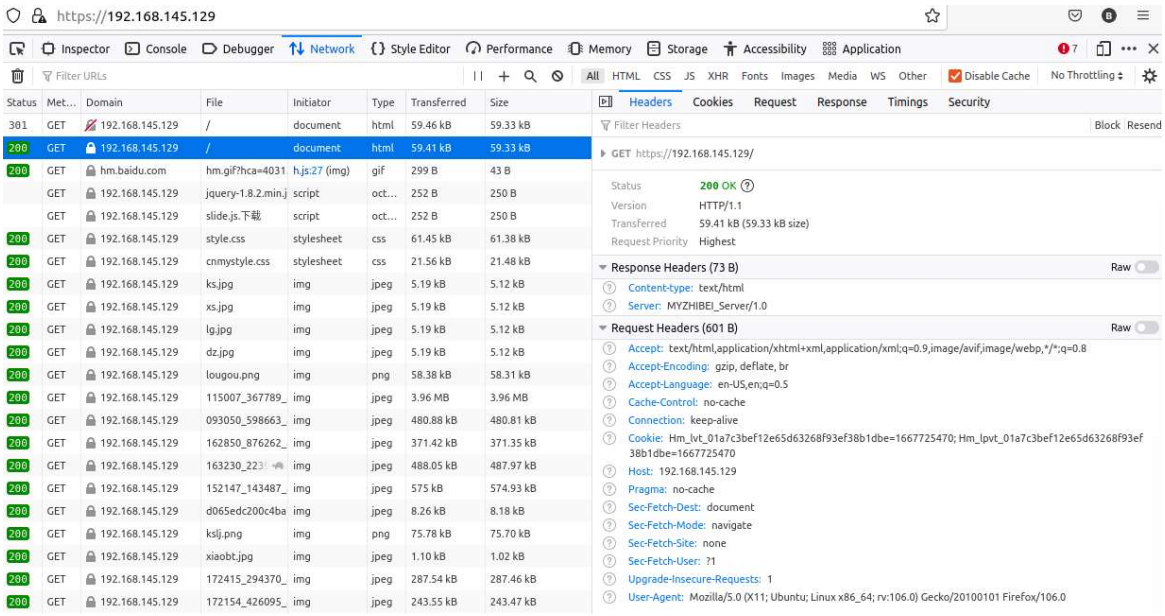

3、测试代码

首先在 Ubuntu 实际环境测试代码。

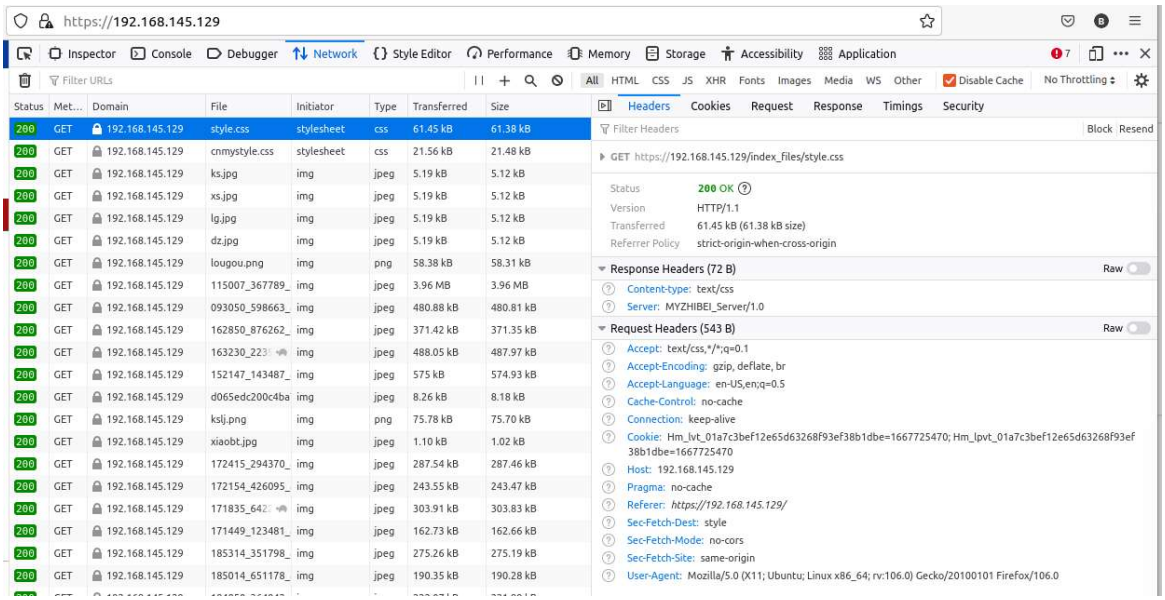
保存好网页文件到服务端根目录，使用浏览器访问本机 IP 地址



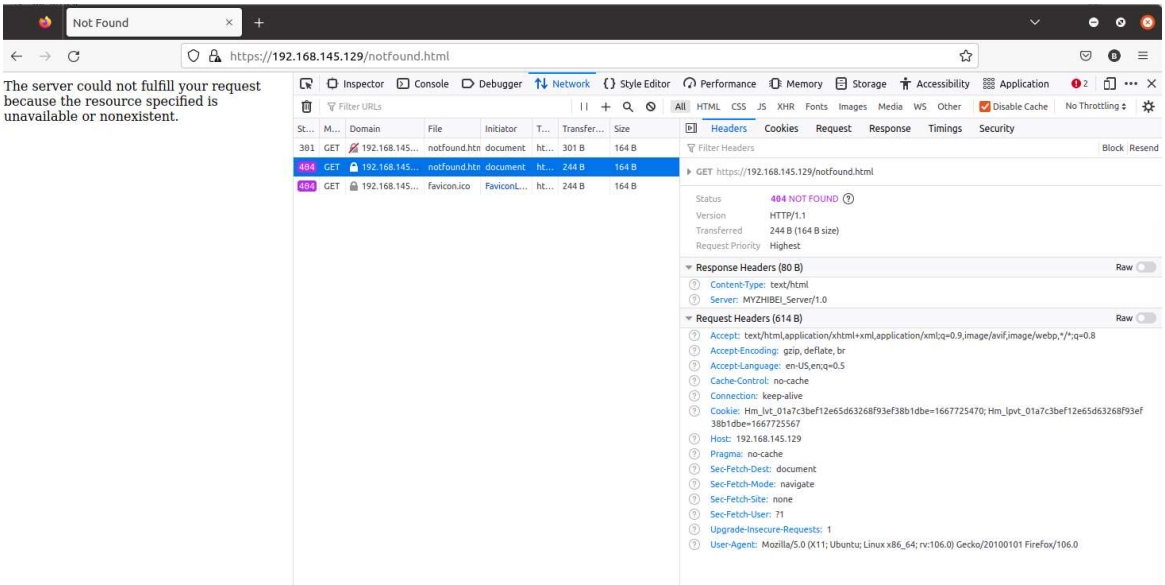

301 跳转正常



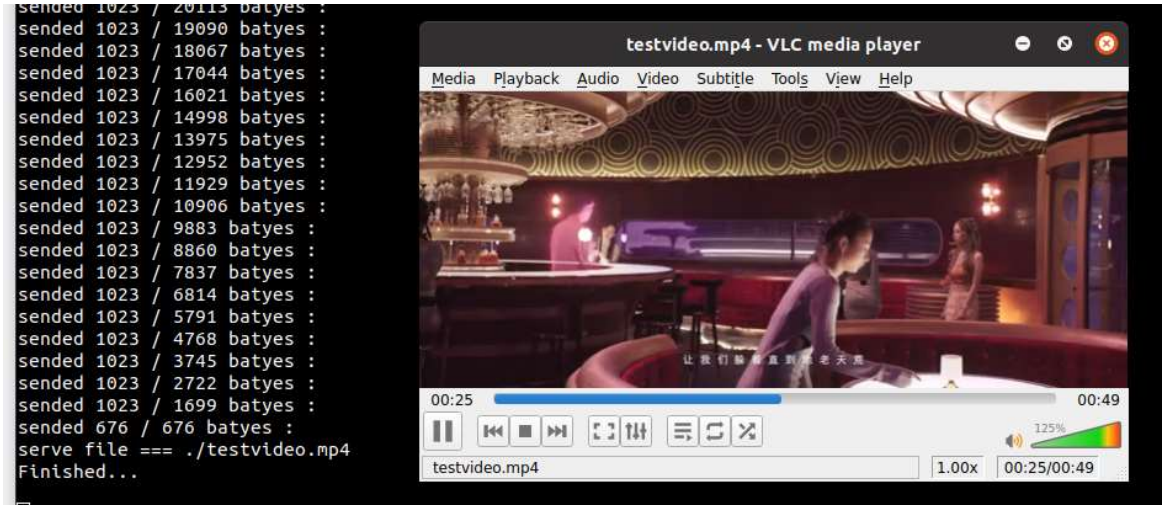

200 GET 正常




文件夹内文件 GTE 200 正常




404 notfound 正常



VLC 播放 https://192.168.145.129/testvideo.mp4 正常

在 mininet 环境中测试

topo.py 内容为

```
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.topo import Topo
from mininet.node import OVSBridge

class MyTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        s1 = self.addSwitch('s1')

        self.addLink(h1, s1, bw=100, delay='10ms')
        self.addLink(h2, s1)

topo = MyTopo()
net = Mininet(topo = topo, switch = OVSBridge, link = TCLink,
controller = None)
net.start()
CLI(net)
net.stop()
```

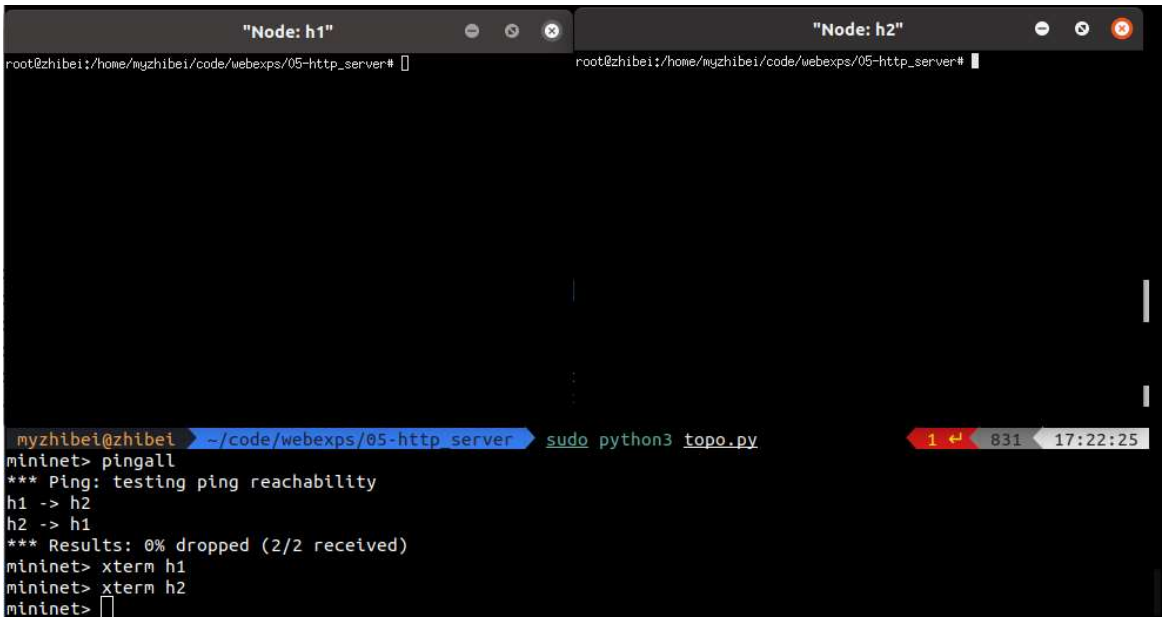
执行

sudo python3 topo.py

pingall

xterm h1

xterm h2



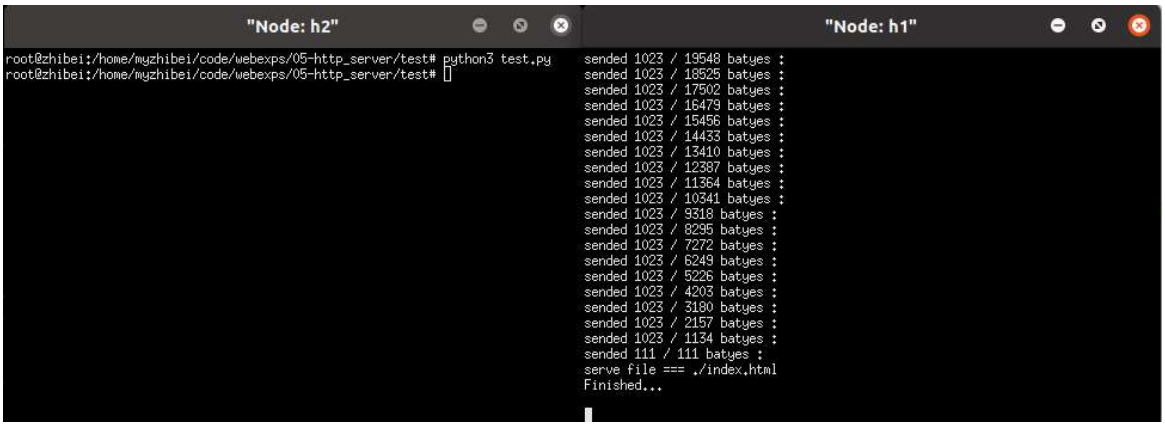

在 h1 中运行 httpServer


在 h2 中运行 test.py

test.py 文件内容为

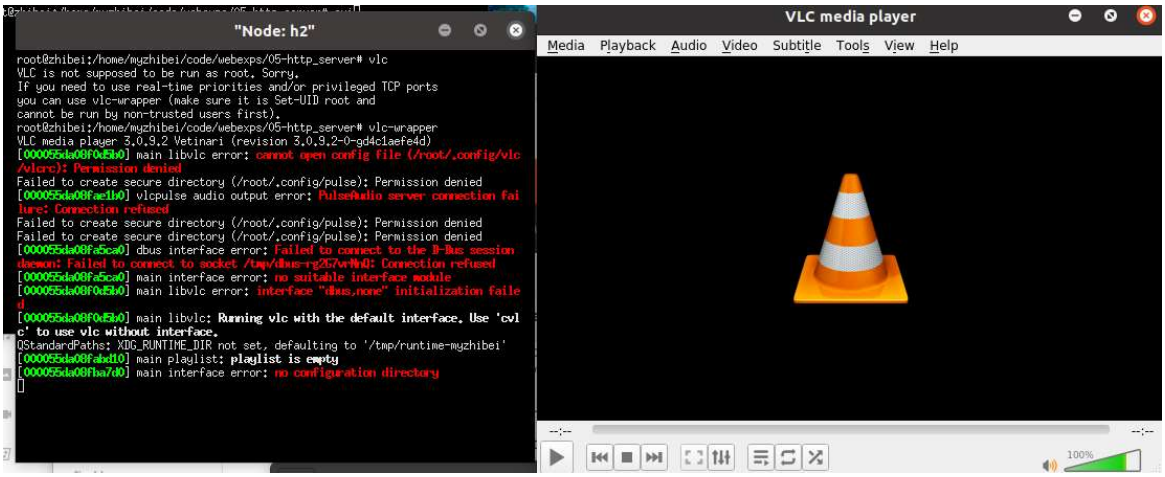
```
import requests
from os.path import dirname, realpath

requests.packages.urllib3.disable_warnings()

test_dir = dirname(realpath(__file__))

# http 301
r = requests.get('http://10.0.0.1/index.html',
allow_redirects=False)
assert(r.status_code == 301 and r.headers['Location'] ==
'https://10.0.0.1/index.html')
```

```
# https 200 OK
r = requests.get('https://10.0.0.1/index.html', verify=False)
assert(r.status_code == 200 and open(test_dir + '/../index.html',
'rb').read() == r.content)

# http 200 OK
r = requests.get('http://10.0.0.1/index.html', verify=False)
assert(r.status_code == 200 and open(test_dir + '/../index.html',
'rb').read() == r.content)

# http 404
r = requests.get('http://10.0.0.1/notfound.html', verify=False)
assert(r.status_code == 404)

# file in directory
r = requests.get('http://10.0.0.1/dir/index.html', verify=False)
assert(r.status_code == 200 and open(test_dir + '/../index.html',
'rb').read() == r.content)

# http 206
headers = { 'Range': 'bytes=100-200' }
r = requests.get('http://10.0.0.1/index.html', headers=headers,
verify=False)
assert(r.status_code == 206 and open(test_dir + '/../index.html',
'rb').read()[100:201] == r.content)

# http 206
headers = { 'Range': 'bytes=100-' }
r = requests.get('http://10.0.0.1/index.html', headers=headers,
verify=False)
assert(r.status_code == 206 and open(test_dir + '/../index.html',
'rb').read()[100:] == r.content)
```



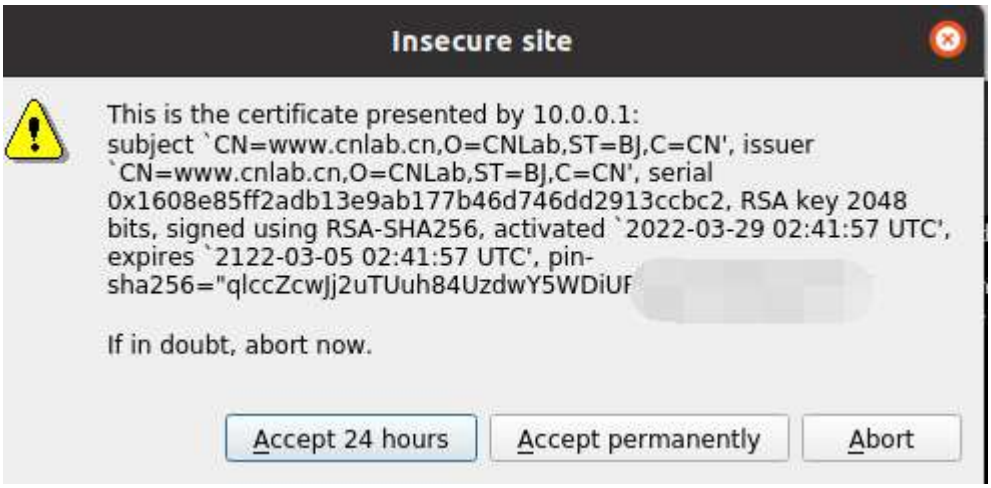

test.py 通过

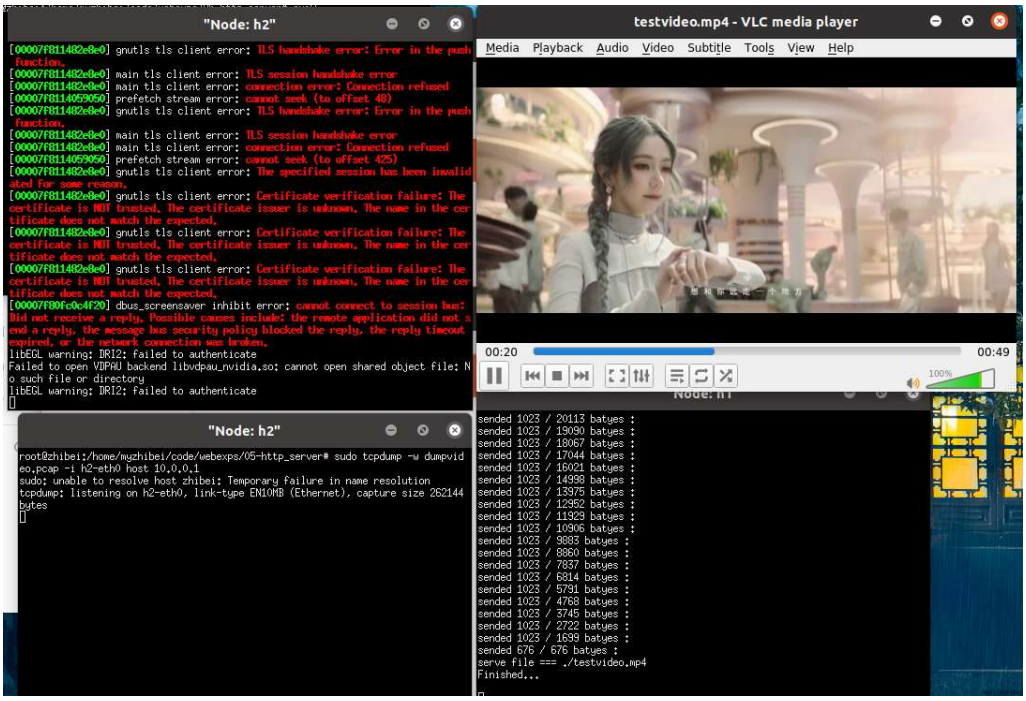
h2 运行 vlc-wrapper
vlc-wrapper




,
再次 xterm h2，在新窗口中 h2 进行抓包
sudo tcpdump -w dumpvideo.pcap -i h2-eth0


然后在 vlc 中打开网络地址 http://10.0.0.1/testvideo.mp4
信任证书


Insecure site
This is the certificate presented by 10.0.0.1:
subject `CN=www.cnlab.cn,O=CNLab,ST=BJ,C=CN', issuer `CN=www.cnlab.cn,O=CNLab,ST=BJ,C=CN', serial 0x1608e85ff2adb13e9ab177b46d746dd2913ccbc2, RSA key 2048 bits, signed using RSA-SHA256, activated `2022-03-29 02:41:57 UTC', expires `2122-03-05 02:41:57 UTC', pin-sha256="qlccZcwJj2uTUuh84UzdwY5WDiUF
If in doubt, abort now.
Accept 24 hours
Accept permanently
Abort

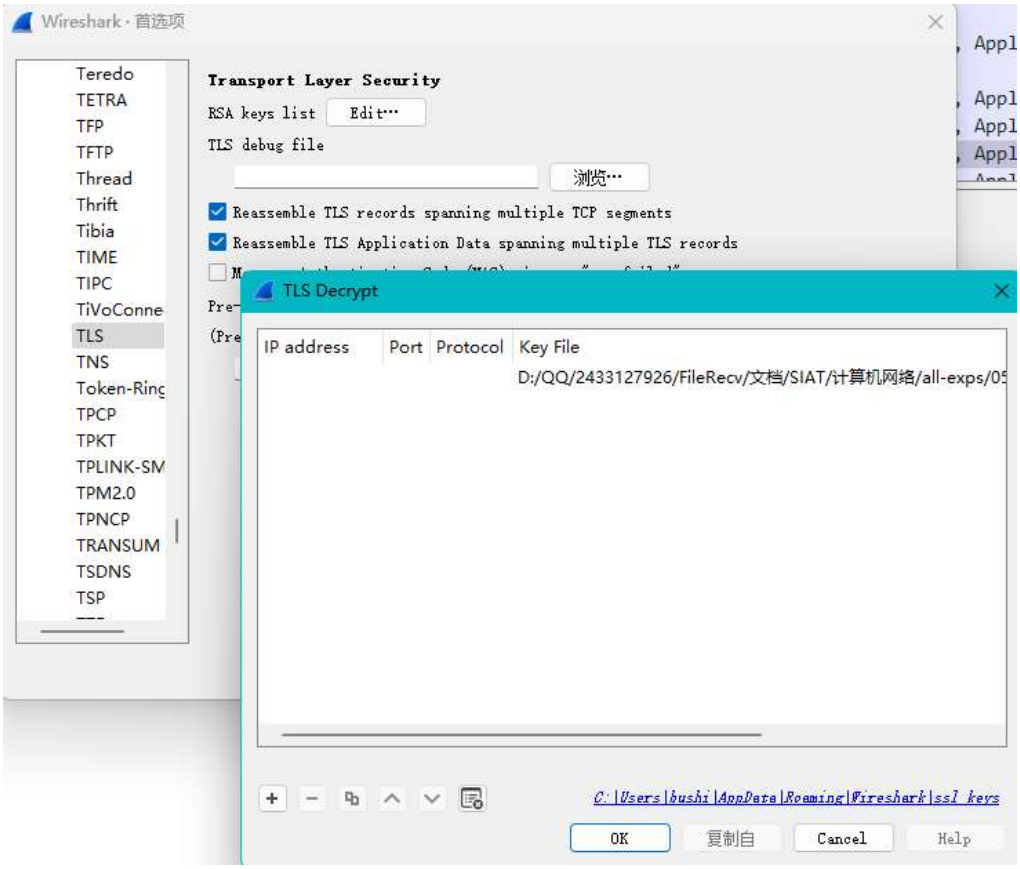

"Node: h2"
testvideo.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
00:20
00:49
"Node: h2"
root@zhibei:/home/myzhibei/code/webexps/05-http_server# sudo tcpdump -w dumpvideo.pcap -i h2-eth0 host 10.0.0.1
sudo: unable to resolve host zhibei: Temporary failure in name resolution
tcpdump: listening on h2-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes

播放成功，结束抓包。

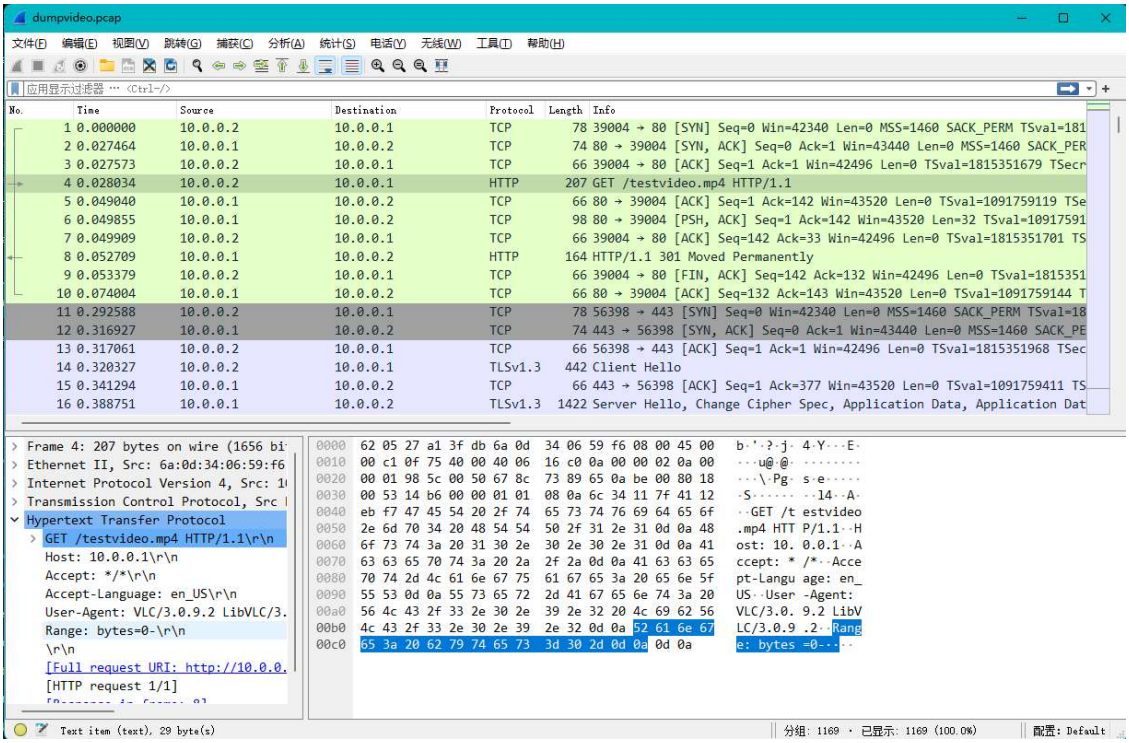

"Node: h2"
root@zhibei:/home/myzhibei/code/webexps/05-http_server# sudo tcpdump -w dumpvideo.pcap -i h2-eth0 host 10.0.0.1
sudo: unable to resolve host zhibei: Temporary failure in name resolution
tcpdump: listening on h2-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C4740 packets captured
4740 packets received by filter
0 packets dropped by kernel
root@zhibei:/home/myzhibei/code/webexps/05-http_server#

将抓包文件用 wireshark 打开进行分析

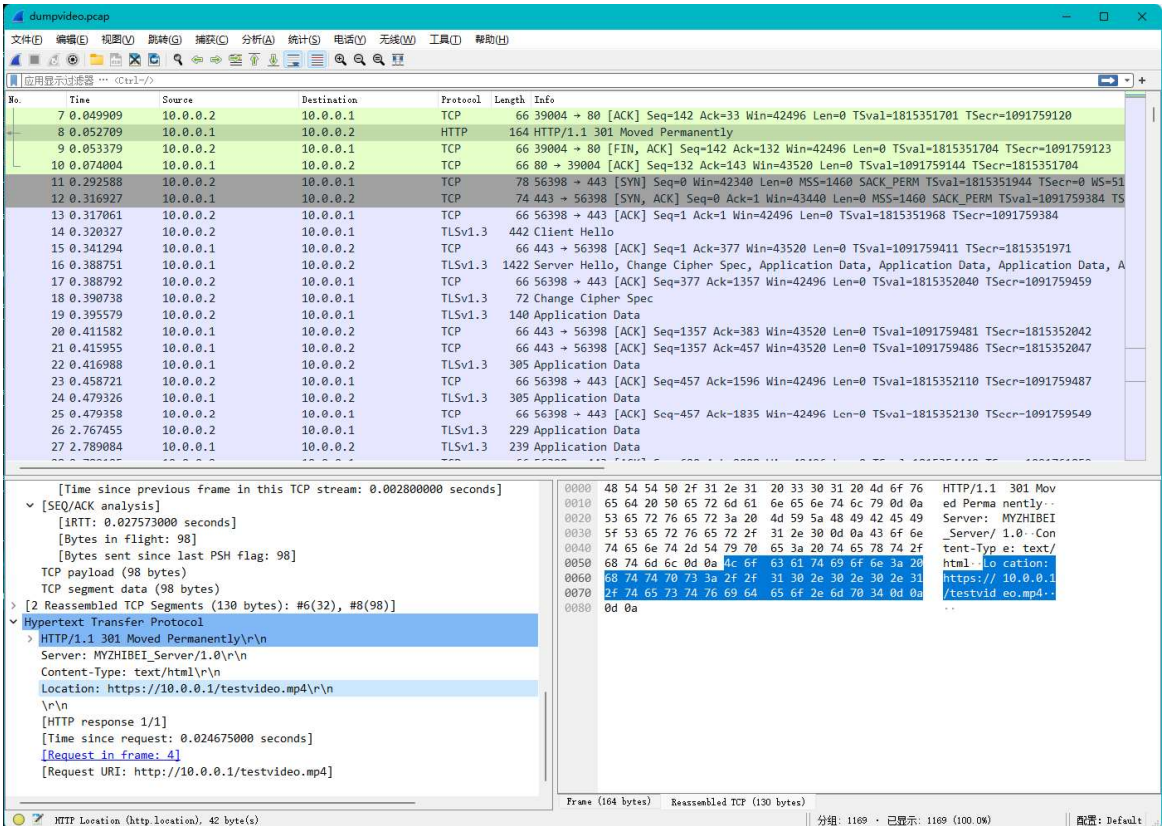
wireshark 加载 TLS 私钥文件




打开 dumpvideo.pcap



可以看到首先是 TCP 三次握手连接，然后是 HTTP 请求获取文件 GET /testvideo.mp4 HTTP/1.1\r\n，其后有一个字段为 Range: bytes=0-，表示请求从 0 字节到文件结束。




服务端返回了 301 告知其 https 地址为 https://10.0.0.1/testvideo.mp4，结束当前连接，之后请求 https，建立 TLS 连接，服务端根据 range 字段发送文件的部分内容，在响应头字段中状态为 206，含有 content-size 字段表示该次发送的为哪个部分。vlc 客户端缓冲一部分，在 TCP windows full 后，vlc 客户端再次发送一个 GET 请求，其 range 字段为 Range: bytes=2571880-，请求后面的部分，手动点击在开始某个部分，关闭后重新打开该视频，客户端默认回到上次观看位置，发送的 range 字段为 Range: bytes=48-，也就是说，vlc 视频流是通过使用 range 字段来发送需要播放的部分来进行缓冲实现在线播放。

## 六、 实验总结

使用 C 语言实现了 HTTP 服务器在 80 和 443 端口多线程同时监听，实现 GET 方法，并将 http 请求转到 https 请求，实现响应状态码 200、206、301、400、404、501，对 Range 字段解析并发送部分文件，对 vlc 打开网络视频流进行抓包分析，加深了对 HTTP 服务器的理解。