

Socket应用编辑实验

学号：2022K8009929011

姓名：王泽黎

一、实验任务

- 实现：使用C语言实现最简单的HTTP服务器
 - 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
 - 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

- 根据上述要求，实现HTTP服务器程序
- 执行`sudo python topo.py`命令，生成包括两个端节点的网络拓扑
- 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`
- 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

三、实验结果与分析

(一) HTTP服务器设计

```
1  int main()
2  {
3      pthread_t thread1, thread2;
4
5      if (pthread_create(&thread1, NULL, HTTP_SERVER, NULL) != 0)
6      {
7          perror("Thread creation failed");
8          return -1;
9      }
10
11     if (pthread_create(&thread2, NULL, HTTPS_SERVER, NULL) != 0)
12     {
13         perror("Thread creation failed");
14         return -1;
15     }
16
17     pthread_join(thread1, NULL);
18     pthread_join(thread2, NULL);
19
20     return 0;
21 }
```

这段代码的作用是创建两个线程，分别执行 HTTP_SERVER 和 HTTPS_SERVER 函数，并在主线程中等待这两个线程完成。通过这种方式，可以同时处理 HTTP 和 HTTPS 请求，实现简单的并发服务器功能。

其中HTTP_SERVER用于支持HTTP，HTTPS_SERVER用于支持HTTPS，下面是对这两个函数的具体介绍

(1) HTTP_SERVER

HTTP 服务器首先是建立 socket 文件描述符，然后绑定监听地址，进而对80接口进行监听，循环不断接收连接请求，在每收到一个服务器的连接后，创建一个新的线程用于接收并解析其HTTP请求，然后根据请求内容进行应答。

```

1 void *HTTP_SERVER(void *arg)
2 {
3     int port = 80; // 定义服务器监听的端口号为 80
4
5     int sockfd;
6     // 创建套接字, 使用 IPv4 地址族(AF_INET)和面向连接的 TCP 协议(SOCK_STREAM)
7     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8     {
9         perror("Create socket failed"); // 创建套接字失败, 输出错误信息
10        exit(1); // 退出程序
11    }
12
13    struct sockaddr_in server;
14    server.sin_family = AF_INET; // 设置地址族为 IPv4
15    server.sin_addr.s_addr = INADDR_ANY; // 监听所有本地 IP 地址
16    server.sin_port = htons(port); // 设置端口号, 使用 htons 将主机字节序转换为网络字节序
17
18    // 绑定套接字到指定的 IP 地址和端口号
19    if (bind(sockfd, (struct sockaddr *)&server, sizeof(server)) < 0)
20    {
21        perror("bind failed"); // 绑定失败, 输出错误信息
22        exit(1); // 退出程序
23    }
24
25    Listen(sockfd, 128); // 将套接字设置为监听模式, 允许最多 128 个待处理连接
26
27    while (1) // 无限循环, 持续接受和处理客户端连接
28    {
29        struct sockaddr_in c_addr; // 定义客户端地址结构
30        socklen_t addr_len; // 定义地址长度变量
31
32        // 接受传入的连接请求
33        int request = accept(sockfd, (struct sockaddr *)&c_addr, &addr_len);
34        if (request < 0)
35        {
36            perror("Accept failed"); // 接受连接失败, 输出错误信息
37            exit(1); // 退出程序
38        }
39
40        pthread_t new_thread; // 定义新线程变量
41
42        // 创建新线程处理客户端请求, 线程函数为 handle_http_request, 参数为请求套接字描述符
43        if ((pthread_create(&new_thread, NULL, (void *)handle_http_request, (void *)&request)) != 0)
44        {
45            perror("Create handle_http_request thread failed"); // 创建线程失败, 输出错误信息
46            exit(1); // 退出程序
47        }
48    }
49
50    close(sockfd); // 关闭套接字 (这行代码在实际运行中不会被执行, 因为 while(1) 是无限循环)
51    return NULL; // 返回 NULL (这行代码在实际运行中不会被执行, 因为 while(1) 是无限循环)
52 }

```

其中 handle_http_request 函数用于处理 HTTP 请求。

首先是接受客户端的request报文, 并检查是否为GET请求。




```

1 // 从请求中读取数据到接收缓冲区
2 request_len = recv(request, recv_buff, 2000, 0);
3 if (request_len < 0)
4 {
5     fprintf(stderr, "recv failed\n"); // 读取失败
6     exit(1);
7 }
8
9 // 检查是否为GET请求
10 char *req_get = strstr(recv_buff, "GET");

```

然后根据request 得到相应的 https URL（这里考虑了是相对的 URL 还是绝对的 URL），最后返回301 Moved Permanently，并在应答中使用 Location 字段表达相应的 https URL。



```

1 // 构建301重定向响应
2 memset(send_buff, 0, 6000);
3 strcat(send_buff, http_version);
4 strcat(send_buff, " 301 Moved Permanently\r\nLocation: ");
5 strcat(send_buff, "https://");

```

从 request 中获取 URL，若为相对路径，则获取Host字段后的信息，然后与相对路径的URL进行拼接，若为绝对路径，则直接写入响应信息。最后将生成的服务器响应发送给客户端即可。



```
1  int i;
2  // 提取URL
3  for (i = 0; (*iterator) != ' '; iterator++, i++)
4  {
5      temp_url[i] = *iterator;
6  }
7  temp_url[i] = '\0';
8  iterator++;
```



```
1 // 如果是相对路径, 提取主机名
2 if (relative_url)
3 {
4     iterator = strstr(recv_buff, "Host:");
5     if(!iterator){
6         perror("Not found Host"); // 未找到Host头
7         exit(1);
8     }
9     iterator += 6; // 跳过 "Host: "
10
11     for (int i = 0; (*iterator) != '\r'; iterator++, i++)
12     {
13         host[i] = *iterator;
14     }
15     host[i] = '\0';
16 }
```




```
1  if (relative_url)
2  {
3      strcat(send_buff, host);
4      strcat(send_buff, temp_url);
5  }
6  else
7  {
8      strcat(send_buff, &temp_url[7]); // 跳过 "http://"
9  }
10 strcat(send_buff, "\r\n\r\n\r\n\r\n");
11
12 // 发送重定向响应
13 if ((send(request, send_buff, strlen(send_buff), 0)) < 0)
14 {
15     fprintf(stderr, "send failed"); // 发送失败
16     exit(1);
17 }
```

(2) HTTPS_SERVER

基本逻辑与HTTP_SERVER类似，使用OpenSSL库来进行加密通信。



```
1 void *HTTPS_SERVER(void *arg)
2 {
3     int port = 443; // 定义服务器监听的端口号为443
4
5     // 初始化SSL库
6     SSL_library_init();
7     OpenSSL_add_all_algorithms();
8     SSL_load_error_strings();
9
10    // 创建SSL方法和上下文
11    const SSL_METHOD *method = TLS_server_method();
12    SSL_CTX *ctx = SSL_CTX_new(method);
13
14    // 加载证书和私钥
15    if (SSL_CTX_use_certificate_file(ctx, "./keys/cnlab.cert", SSL_FILETYPE_PEM) <= 0)
16    {
17        perror("load cert failed"); // 加载证书失败
18        exit(1);
19    }
20    if (SSL_CTX_use_PrivateKey_file(ctx, "./keys/cnlab.prikey", SSL_FILETYPE_PEM) <= 0)
21    {
22        perror("load prikey failed"); // 加载私钥失败
23        exit(1);
24    }
25
26    int sockfd;
27    // 创建套接字
28    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
29    {
30        perror("Create socket failed"); // 创建套接字失败
31        exit(1);
32    }
33
34    struct sockaddr_in server;
35    server.sin_family = AF_INET; // 使用IPv4地址
36    server.sin_addr.s_addr = INADDR_ANY; // 绑定到所有可用的接口
37    server.sin_port = htons(port); // 设置端口号
38
39    // 绑定套接字到端口
40    if (bind(sockfd, (struct sockaddr *)&server, sizeof(server)) < 0)
41    {
42        perror("bind failed"); // 绑定失败
43        exit(1);
44    }
45
46    // 开始监听连接请求
47    listen(sockfd, 10);
48
49    while (1)
50    {
51        struct sockaddr_in c_addr;
52        socklen_t addr_len;
53
54        // 接受客户端连接请求
55        int request = accept(sockfd, (struct sockaddr *)&c_addr, &addr_len);
56        if (request < 0)
57        {
58            perror("Accept failed"); // 接受连接失败
59            exit(1);
60        }
61    }
62}
```

```

60     }
61
62     // 创建新的SSL对象并将其与请求的文件描述符关联
63     SSL *ssl = SSL_new(ctx);
64     SSL_set_fd(ssl, request);
65
66     pthread_t new_thread;
67
68     // 创建新线程处理HTTPS请求
69     if ((pthread_create(&new_thread, NULL, (void *)handle_https_request, (void *)ssl)) != 0)
70     {
71         perror("Create handle_https_request thread failed"); // 创建线程失败
72         exit(1);
73     }
74 }
75
76 // 关闭套接字并释放SSL上下文
77 close(sockfd);
78 SSL_CTX_free(ctx);
79 return NULL;
80 }

```

其中 `handle_https_request` 函数用于处理 HTTPS 请求。


首先第一步是执行SSL握手，然后由于客户端可能需要继续保持连接，这里设置了一个变量（keep-alive）用于处理需要继续保持连接的情况。之后开始接受客户端的request报文，并检查是否为GET请求，然后对请求进行解析，得到请求的相关信息。而根据解析的信息，确定要发送的文件路径。如果文件不存在，发送一个404错误响应，否则看是否有Range字段，如果有Range字段，返回206 Partial Content，若没有会返回200 OK。而如果文件存在，会根据请求返回文件的对应部分的内容。

SSL握手：



```
1  // 进行 SSL 握手
2  if (SSL_accept(ssl) == -1)
3  {
4      perror("SSL_accept failed"); // 握手失败
5      exit(1);
6  }
```

Range字段的解析：



```
1 // 检查是否为范围请求
2 if((iterator = strstr(recv_buff, "Range:")))
3 {
4     iterator += 13;
5     range = 1;
6
7     range_begin = 0;
8     while(*iterator >= '0' && *iterator <= '9')
9     {
10         range_begin = range_begin * 10 + (*iterator) - '0';
11         iterator++;
12     }
13     iterator++;
14
15     if(*iterator < '0' || *iterator > '9')
16     {
17         range_end = -1;
18     }
19     else
20     {
21         range_end = 0;
22         while(*iterator >= '0' && *iterator <= '9')
23         {
24             range_end = range_end * 10 + (*iterator) - '0';
25             iterator++;
26         }
27     }
28 }
```

判断是否需要继续保持连接:



```
1 // 检查连接类型
2 if((iterator = strstr(recv_buff, "Connection:"))
3 {
4     iterator += 12;
5     if(*iterator == 'k'){
6         keep_alive = 1; // 保持连接
7     }
8     else if(*iterator == 'c'){
9         keep_alive = 0; // 关闭连接
10    }
11 }
```

(二) 实验结果

```
zeri@zeri-virtual-machine: ~/2024Fall_CNLab/03-Socket
zeri@zeri-virtual-machine:~/2024Fall_CNLab/03-Socket$ make
gcc -Wall -g *.c -o http-server -lssl -lcrypto -lpthread
zeri@zeri-virtual-machine:~/2024Fall_CNLab/03-Socket$ sudo python topo.py
[sudo] password for zeri:
sudo: python: command not found
zeri@zeri-virtual-machine:~/2024Fall_CNLab/03-Socket$ sudo python3 topo.py
mininet> xterm h1 h2
mininet>

"Node: h1"
root@zeri-virtual-machine:/home/zeri/2024Fall_CNLab/03-Socket# ./http-server

"Node: h2"
root@zeri-virtual-machine:/home/zeri/2024Fall_CNLab/03-Socket# python3 test/test.py
root@zeri-virtual-machine:/home/zeri/2024Fall_CNLab/03-Socket#
```

四、实验总结

通过本次实验，我对Socket API有了一定的了解。Socket API对上层提供统一的调用接口，提供最基本的网络通信功能。通过实际编写一个简单的HTTP服务器，我对于建立Socket描述符、建立连接等内容以及HTTPS加密通信都有了不少了解，对于客户端、服务器之间的交互机制也有了一定的认识。此外我也学到了HTTP协议的内容，主要是对HTTP报文的结构有了不少了解，对HTTPS与HTTP的区别也有了更深的认识，因此本次实验让我对网络文件传输有了更深的理解。