浙江大学

本科实验报告

课程名称：　计算机网络基础

实验名称：　实现一个轻量级的 WEB 服务器

姓　　名：　何煦明

学　　院：　计算机学院

系：

专　　业：　计算机科学与技术

学　　号：　3210101822

指导教师：　陆系群

2023 年 12 月 18 日

浙江大学实验报告

实验名称：实现一个轻量级的 WEB 服务器　实验类型：编程实验

同组学生：无　实验地点：计算机网络实验室

一、 实验目的

深入掌握 HTTP 协议规范，学习如何编写标准的互联网应用服务器。

二、 实验内容

- 服务程序能够正确解析 HTTP 协议，并传回所需的网页文件和图片文件
- 使用标准的浏览器，如 IE、Chrome 或者 Safari，输入服务程序的 URL 后，能够正常显示服务器上的网页文件和图片
- 服务端程序界面不做要求，使用命令行或最简单的窗体即可
- 功能要求如下：
  1. 服务程序运行后监听在 80 端口或者指定端口
  2. 接受浏览器的 TCP 连接（支持多个浏览器同时连接）
  3. 读取浏览器发送的数据，解析 HTTP 请求头部，找到感兴趣的部分
  4. 根据 HTTP 头部请求的文件路径，打开并读取服务器磁盘上的文件，以 HTTP 响应格式传回浏览器。要求按照文本、图片文件传送不同的 Content-Type，以便让浏览器能够正常显示。
  5. 分别使用单个纯文本、只包含文字的 HTML 文件、包含文字和图片的 HTML 文件进行测试，浏览器均能正常显示。
- 本实验可以在前一个 Socket 编程实验的基础上继续，也可以使用第三方封装好的 TCP 类进行网络数据的收发
- 本实验要求不使用任何封装 HTTP 接口的类库或组件，也不使用任何服务端脚本程序如 JSP、ASPX、PHP 等

三、 主要仪器设备

联网的 PC 机、Wireshark 软件、Visual Studio、gcc 或 Java 集成开发环境。

四、 操作方法与实验步骤

- 阅读 HTTP 协议相关标准文档，详细了解 HTTP 协议标准的细节，有必要的话使用 Wireshark 抓包，研究浏览器和 WEB 服务器之间的交互过程
- 创建一个文档目录，与服务器程序运行路径分开
- 准备一个纯文本文件，命名为 test.txt，存放在 txt 子目录下面
- 准备好一个图片文件，命名为 logo.jpg，放在 img 子目录下面
- 写一个 HTML 文件，命名为 test.html，放在 html 子目录下面，主要内容为：

```
<html>
    <head><title>Test</title></head>
    <body>
        <h1>This is a test</h1>
        <img src="img/logo.jpg">
        <form action="dopost" method="POST">
            Login:<input name="login">
            Pass:<input name="pass">
            <input type="submit" value="login">
        </form>
    </body>
</html>
```

- 将 test.html 复制为 noimg.html，并删除其中包含 img 的这一行。
- 服务端编写步骤（**需要采用多线程模式**）

a) 运行初始化，打开 Socket，监听在指定端口（**请使用学号的后 4 位作为服务器的监听端口**）

b) 主线程是一个循环，主要做的工作是等待客户端连接，如果有客户端连接成功，为该客户端创建处理子线程。该子线程的主要处理步骤是：

1. 不断读取客户端发送过来的字节，并检查其中是否连续出现了 2 个回车换行符，如果未出现，继续接收；如果出现，按照 HTTP 格式解析第 1 行，分离出方法、文件和路径名，其他头部字段根据需要读取。

✧ **如果解析出来的方法是 GET**

2. 根据解析出来的文件和路径名，读取响应的磁盘文件（该路径和服务器程序可能不在同一个目录下，需要转换成绝对路径）。如果文件不存在，第 3 步的响应消息的状态设置为 404，并且跳过第 5 步。
3. 准备好一个足够大的缓冲区，按照 HTTP 响应消息的格式先填入第 1 行（状态码=200），加上回车换行符。然后模仿 Wireshark 抓取的 HTTP 消息，填入必要的几行头部（需要哪些头部，请试验），其中不能缺少的 2 个头部是 Content-Type 和 Content-Length。Content-Type 的值要和文件类型相匹配（请通过抓包确定应该填什么），Content-Length 的值填写文件的字节大小。
4. 在头部行填完后，再填入 2 个回车换行
5. 将文件内容按顺序填入到缓冲区后面部分。

✧ **如果解析出来的方法是 POST**

6. 检查解析出来的文件和路径名，如果不是 dopost，则设置响应消息的状态为 404，然后跳到第 9 步。如果是 dopost，则设置响应消息的状态为 200，并继续下一步。
7. 读取 2 个回车换行后面的体部内容（长度根据头部的 Content-Length 字段的指示），并提取出登录名（login）和密码（pass）的值。如果登录名是你的学号，密码是学号的后 4 位，则将响应消息设置为登录成功，否则将响应消息设置为登录失败。
8. 将响应消息封装成 html 格式，如

<html><body>响应消息内容</body></html>

9. 准备好一个足够大的缓冲区，按照 HTTP 响应消息的格式先填入第 1 行（根据前面的情况设置好状态码），加上回车换行符。然后填入必要的几行头部，其中不能缺少的 2 个头部是 Content-Type 和 Content-Length。Content-Type 的值设置为 text/html，如果状态码=200，则 Content-Length 的值填写响应消息的字节大小，并将响应消息填入缓冲区的后面部分，否则填写为 0。
10. 最后一次性将缓冲区内的字节发送给客户端。
11. 发送完毕后，关闭 socket，退出子线程。

c) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 Socket，主程序退出。

- 编程结束后，将服务器部署在一台机器上（本机也可以）。在服务器上分别放置纯文本文件（.txt）、只包含文字的测试 HTML 文件（将测试 HTML 文件中的包含 img 那一行去掉）、包含文字和图片的测试 HTML 文件（以及图片文件）各一个。
- 确定好各个文件的 URL 地址，然后使用浏览器访问这些 URL 地址，如 http://x.x.x.x:port/dir/a.html，其中 port 是服务器的监听端口，dir 是提供给外部访问的路径，请设置为与文件实际存放路径不同，通过服务器内部映射转换。
- 检查浏览器是否正常显示页面，如果有问题，查找原因，并修改，直至满足要求
- 使用多个浏览器同时访问这些 URL 地址，检查并发性

五、 实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

- **源代码：需要说明编译环境和编译方法，如果不能编译成功，将影响评分**
- **可执行文件：可运行的.exe 文件或 Linux 可执行文件**

以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- 服务器的主线程循环关键代码截图（解释总体处理逻辑，省略细节部分）

```
while(1){
    clntSock=accept(servSock, (struct sockaddr*)(&clntAddr), &clntAddrSize);
    if(clntSock==-1){
        //建立连接失败
        cout<<"Failed to connect."<<endl;
        cout<<"+----------------+"<<endl;
    }
    else{
        //成功建立连接
        cout<<"Success to connect."<<endl;
        cout<<YELLOW_COLOR<<"Client ip: "<<RESET_COLOR<<decimalToIPv4(clntAddr.sin_addr.s_addr)<<endl;
        cout<<YELLOW_COLOR<<"Client port: "<<RESET_COLOR<<ntohs(clntAddr.sin_port)<<endl;
        //新建服务线程
        pthread_t tid;
        pthread_create(&tid, NULL, &handleRequest, (void*)(&clntSock));
        cout<<"+----------------+"<<endl;
    }
}
```

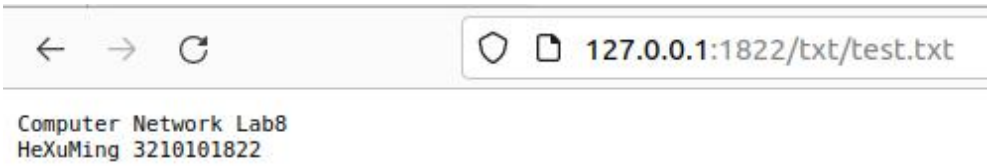
- 服务器的客户端处理子线程关键代码截图（解释总体处理逻辑，省略细节部分）

```
void* handleRequest(void* arg)
{
    int sock=*(int*)arg;
    char method[BUFFER_SIZE];
    char uri[BUFFER_SIZE];
    char header[BUFFER_SIZE];
    char content[BUFFER_SIZE];
    char version[BUFFER_SIZE];
    struct stat FileState;
    ParseHttp(method, uri, version, header, content, sock);
    if(!strcasecmp(method, "GET")){
        ...
    }
    else if(!strcasecmp(method, "POST")){
        ...
    }
    pthread_exit(nullptr);
}
```

- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
tcp        0      0 0.0.0.0:1822            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:52456         127.0.0.1:1822          FIN_WAIT2
tcp        0      0 127.0.0.1:1822          127.0.0.1:41554         TIME_WAIT
tcp        0      0 127.0.0.1:1822          127.0.0.1:52456         CLOSE_WAIT
tcp        0      0 127.0.0.1:41570         127.0.0.1:1822          FIN_WAIT2
tcp        0      0 127.0.0.1:1822          127.0.0.1:41546         CLOSE_WAIT
tcp        0      0 127.0.0.1:41546         127.0.0.1:1822          FIN_WAIT2
tcp        1      0 127.0.0.1:1822          127.0.0.1:41570         CLOSE_WAIT
```

- 浏览器访问纯文本文件（.txt）时，浏览器的 URL 地址和显示内容截图。



服务器上文件实际存放的路径：

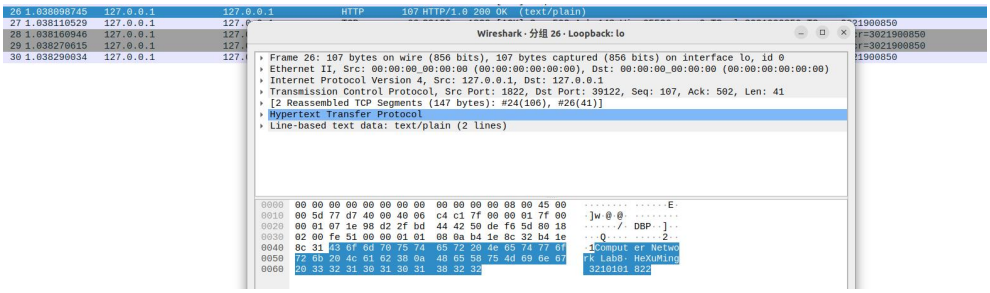
服务器的相关代码片段：

```
if(!strcasecmp(method, "GET")){
    //请求方法为GET
    ConvertFileName(uri);
    if(stat(uri, &FileState)<0){
        //找不到文件
        SendMsg(sock, "404", "Not found");
    }
    else if((S_IRUSR&FileState.st_mode)==0){
        //没有权限
        SendMsg(sock, "403", "Forbidden");
    }
    else{
        //发送响应包
        SendFile(sock, uri, FileState.st_size);
    }
}
```

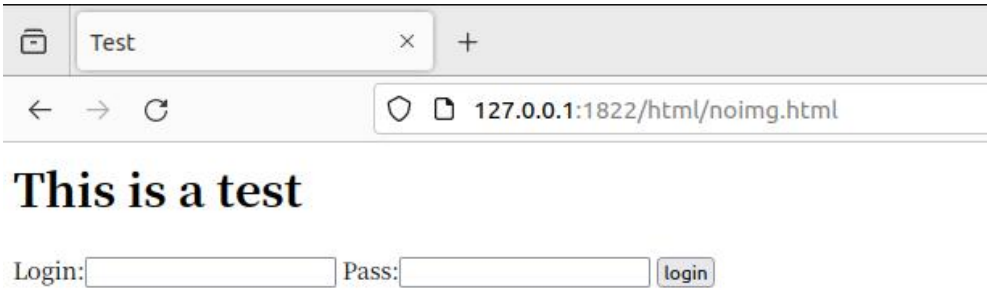
当解析出 GET 命令时，首先判断文件的状态。当文件不存在时发送 404 Not Found 信息，当没有足够访问权限时发送 403 Forbidden 信息。当状态正常时发送数据内容。其中 SendFile 函数代码如下：

```
void SendFile(int fd, char* uri, off_t size)
{
    char* mappedfp;
    char buf[BUFFER_SIZE]={};
    char filetype[BUFFER_SIZE]={};
    GetType(uri, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: A Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %lld\r\n", buf, size);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    //将响应信息的header发给客户端
    write(fd, buf, strlen(buf));
    cout<<YELLOW_COLOR<<"SendFile.header:"<<RESET_COLOR<<endl;
    printf("%s", buf);
    //将响应信息的内容发送给客户端
    int filefd=open(uri, O_RDONLY, 0);
    mappedfp=(char*)mmap(nullptr, size, PROT_READ, MAP_PRIVATE, filefd, 0);
    write(fd, mappedfp, size);
    cout<<YELLOW_COLOR<<"SendFile.content:"<<RESET_COLOR<<endl;
    printf("%s\n", mappedfp);
    //解除内存映射
    munmap(mappedfp, size);
    close(fd);
    cout<<"+----------------+"<<endl;
}
```

Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：

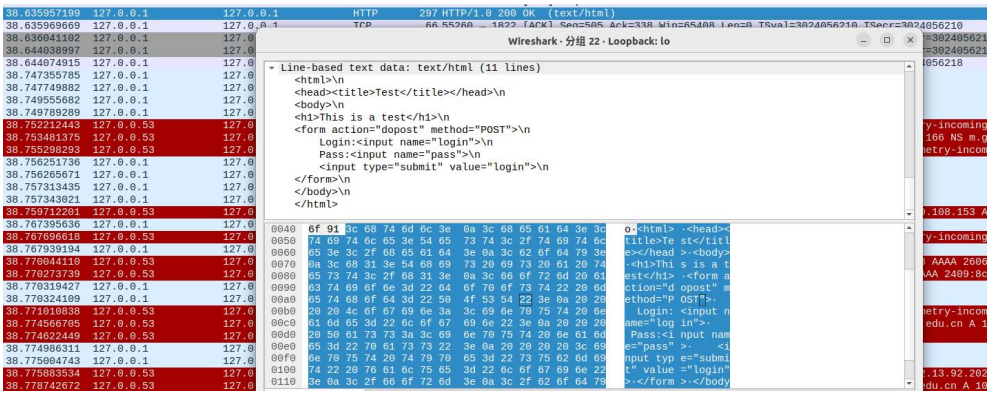


- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。

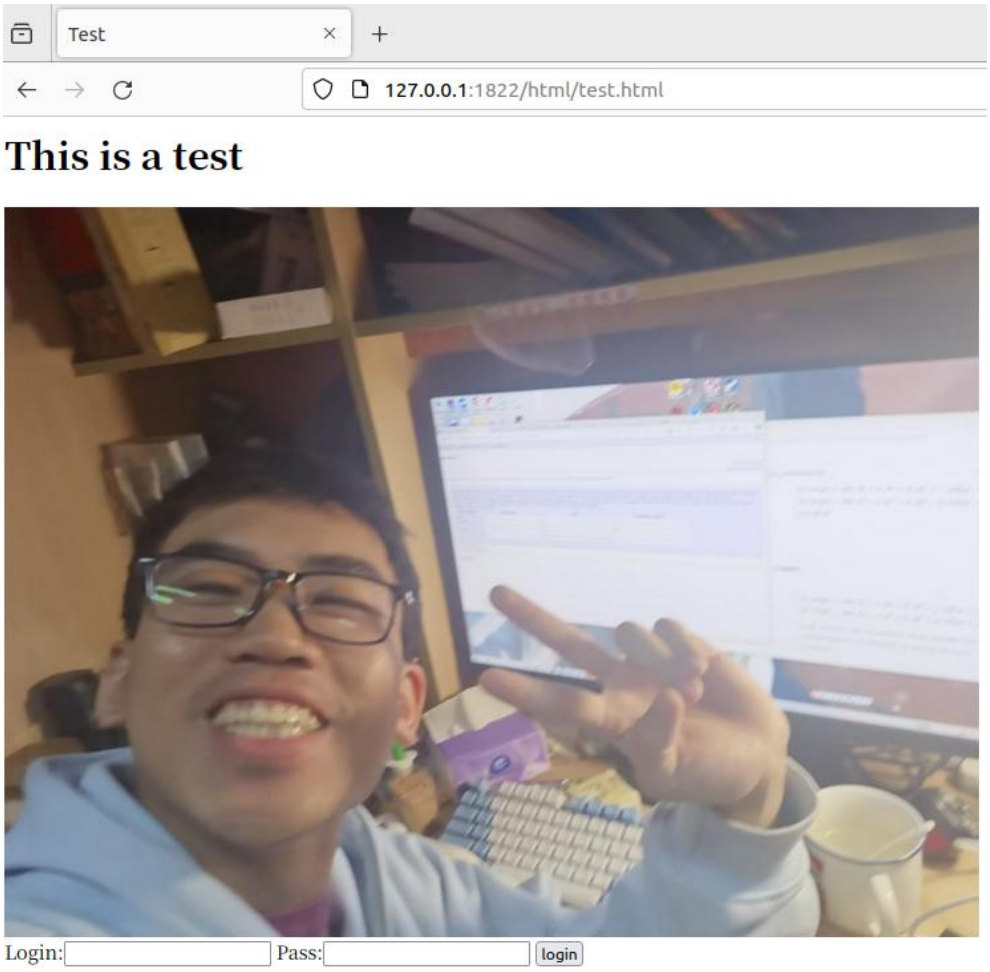


服务器文件实际存放的路径：

Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：



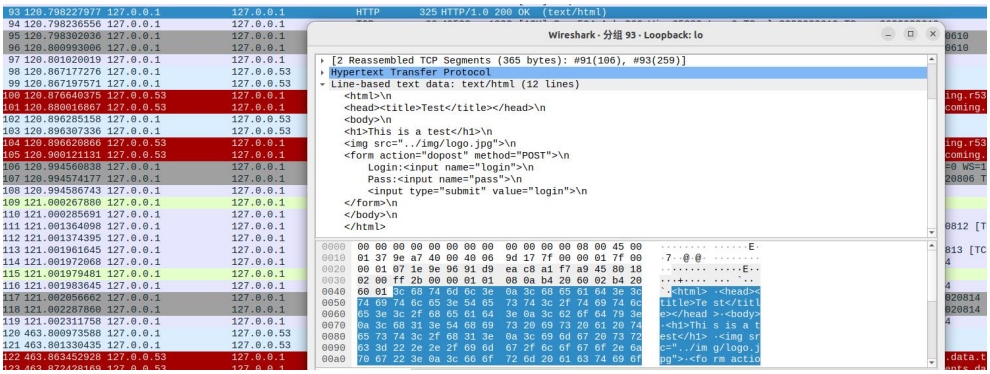
- 浏览器访问包含文本、图片的 HTML 文件时，浏览器的 URL 地址和显示内容截图。



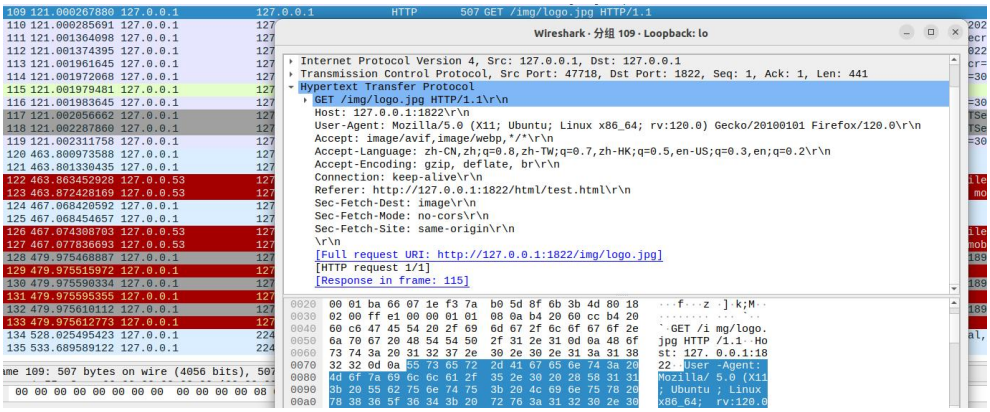
服务器上文件实际存放的路径：

Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

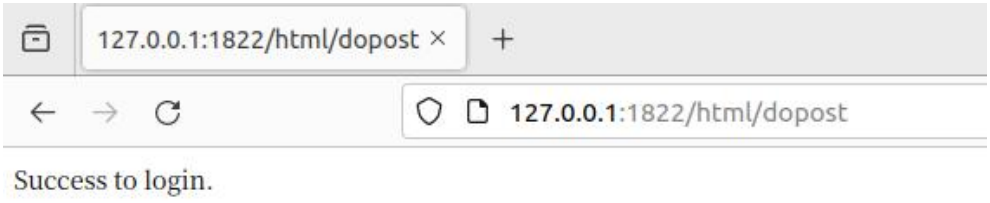
HTML 内容如下：



图片内容如下：



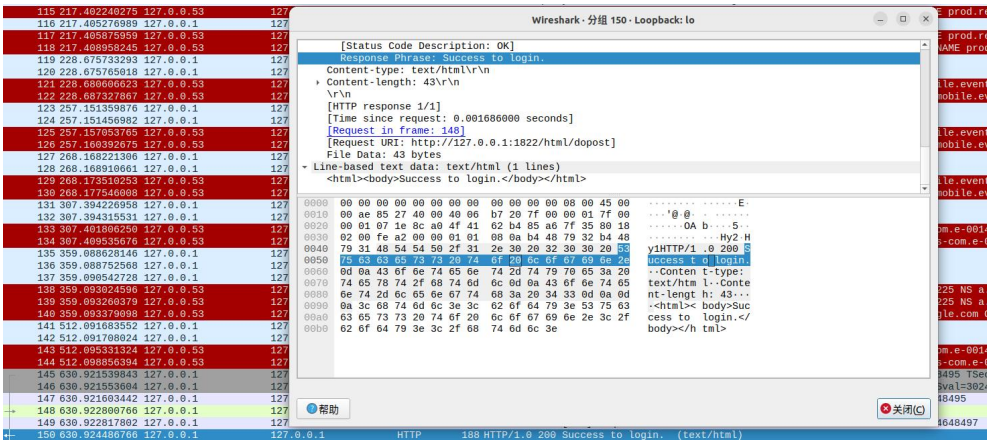
- 浏览器输入正确的登录名或密码，点击登录按钮（login）后的显示截图。



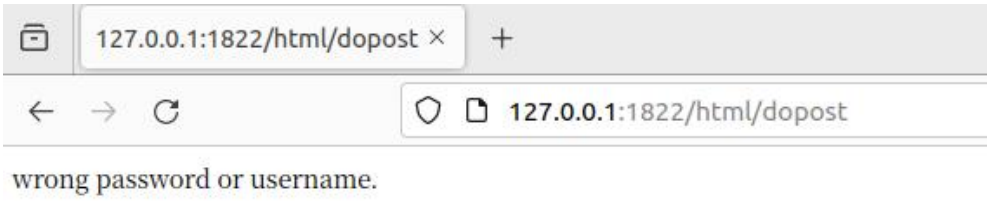
服务器相关处理代码片段：

```
else if(!strcasecmp(method, "POST")){
        //请求方法为POST
        if(strcmp(uri, "/html/dopost")){
            //请求的资源不存在
            SendMsg(sock, "404", "Not found");
        }
        else if(!strcmp(content, "login=3210101822&pass=1822")){
            //登录成功
            SendMsg(sock, "200", "Success to login.");
        }
        else{
            //登录失败
            SendMsg(sock, "200", "wrong password or username.");
        }
    }
```

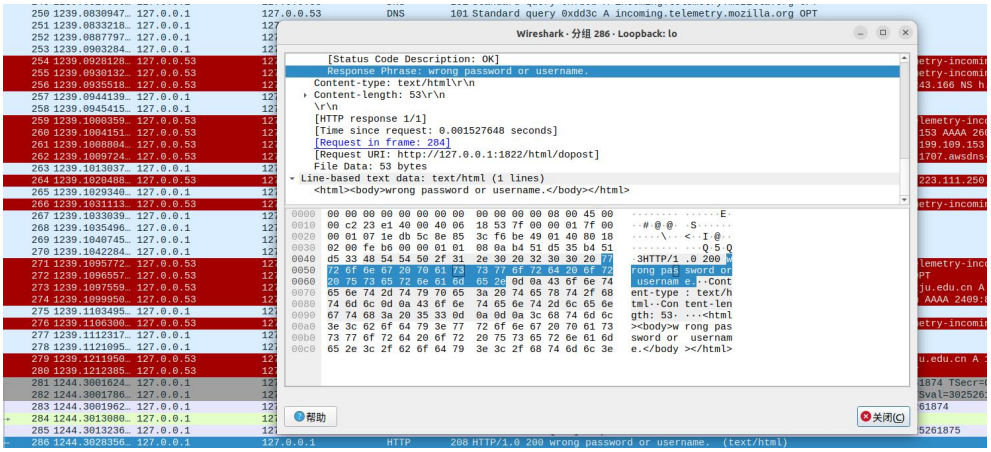
Wireshark 抓取的数据包截图（HTTP 协议部分）



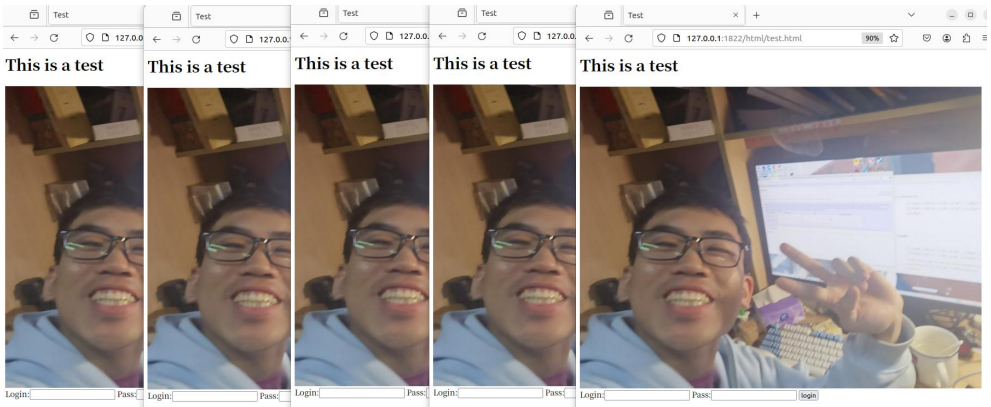
- 浏览器输入错误的登录名或密码，点击登录按钮（login）后的显示截图。



- Wireshark 抓取的数据包截图（HTTP 协议部分）



- 多个浏览器同时访问包含图片的 HTML 文件时，浏览器的显示内容截图（将浏览器窗口缩小并列）



- 多个浏览器同时访问包含图片的 HTML 文件时，使用 netstat –an 显示服务器的 TCP 连接（截取与服务器监听端口相关的）

```
tcp        0      0 0.0.0.0:1822            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:1822          127.0.0.1:41422         CLOSE_WAIT
tcp        0      0 127.0.0.1:1822          127.0.0.1:51060         CLOSE_WAIT
tcp        0      0 127.0.0.1:1822          127.0.0.1:36146         CLOSE_WAIT
tcp        0      0 127.0.0.1:1822          127.0.0.1:52210         CLOSE_WAIT
tcp        1      0 127.0.0.1:1822          127.0.0.1:56156         CLOSE_WAIT
tcp        1      0 127.0.0.1:1822          127.0.0.1:36000         CLOSE_WAIT
tcp        0      0 127.0.0.1:1822          127.0.0.1:44908         CLOSE_WAIT
tcp        1      0 127.0.0.1:1822          127.0.0.1:38132         CLOSE_WAIT
```

六、 实验结果与分析

根据你编写的程序运行效果，分别解答以下问题（看完请删除本句）：

- HTTP 协议是怎样对头部和体部进行分隔的？

请求头：

请求行以方法字段开始，后面分别是 URL 字段和 HTTP 协议版本字段，并以回车符加换行符结尾，之后是一些头部域的名称，后面跟头部域的值。数据和头部之间会有一个“空行”，仅有一个回车符和换行符。

响应头：

响应报文与请求报文类似，以版本字段开始。后面分别是状态码和原因短语。之后是一些头部域的名称，后面跟头部域的值。同样，数据和头部之间会有一个“空行”，包含一个回车和一个换行符，来讲头部和数据分开。

- 浏览器是根据文件的扩展名还是根据头部的哪个字段判断文件类型的？

  根据头部字段中头部域为 Content-type 的内容来判断文件类型。

- HTTP 协议的头部是不是一定是文本格式？体部呢？

  头部是纯文本格式，但是体部不一定，需要根据 Content-type 来确定。本次实验中包含了.jpg 格式。

- POST 方法传递的数据是放在头部还是体部？两个字段是用什么符号连接起来的？

  放在体部，两个字段是用&符号连接起来的。

七、 讨论、心得

实验之前需要先了解 http 协议的报文，需要做一些准备工作。本次实验我是在 VMvare 虚拟机上完成的，由于虚拟机使用的网络适配器是本机的 VMnet1，相当于如果不更改虚拟机设置的话是很难在其它电脑上通过 HTTP 连接到服务端的。但是在本地的物理机上是可以的。同时在使用 Wireshark 抓包时需要选择本地回环抓包(LoopBack)，这也是需要注意的。