计算机网络专题实验 实验八报告

姓名：田濡豪 班级：计算机 2101

姓名：李昱超 班级：信息 2104

1 实验名称

Socket 网络编程实验：基于 HTTP 协议的客户端程序（文本浏览器）

2 实验目的

1)掌握 Sockets 的相关基础知识，学习 Sockets 编程的基本函数和模式、框架。

2)掌握 UDP、TCP 协议及 Client/Server 和 P2P 两种模式的通信原理。掌握可靠数据传输协议（GBN 和 SR 协议）

3)掌握 socket 编程框架

3 实验内容

3.1 基本功能

- 实现 GET、HEAD、POST 三种请求方法
- 实现十六进制替换 URL 中的不安全字符
- 实现可选择的持久连接
- 实现批量获取服务器对象
- 实现基于 Cookie 的网站登陆和 Session 有状态连接
- 实现重定向并可定义重定向/重试次数

- 实现基于 MD5 哈希校验的文件缓存
- 实现常见错误码的处理和用户友好的错误提示

3.2 高级功能

- 实现分块传输编码
- 实现 gzip 内容编码并与分块传输兼容
- 实现基于 POST 方法的批量文件上传
- （额外功能）客户端实现 Socket 传输重试和非阻塞模式数据读取
- （额外功能）客户端实现基于 Textual TUI 的响应式交互界面
- （额外功能）客户端实现服务器文本文件实时预览和代码高亮
- （额外功能）后端 APACHE 服务器实现基于 WSGI 的自定义服务 API

4 实验实现

4.1 人员分工

田濡豪：

- 需求分析与架构设计
- UI 设计
- 前后端代码实现
- 测试用例实现与调试
- 服务器部署
- 部分报告撰写

李昱超：

- 测试用例实现及调试
- 多场景部署测试
- 报文截获及分析
- 部分报告撰写

4.2 实验设计

4.2.1 需求分析与整体架构设计

实验设计首先从模拟一个典型的敏捷开发开始：从实验要求中抽象并提炼用户需求，将这些用户需求组合成不同的典型用例用以描述用户的行为。

比如，如果用户拿到了一个符合实验要求的文本浏览器，一个典型的文件浏览用例如下：

用例 1：基本浏览者

参与者：用户、客户端
前置条件：用户具备可用的互联网连接，客户端已成功安装；服务器可用且可访问。
后置条件：用户可以浏览服务器上的文件。
主流程：

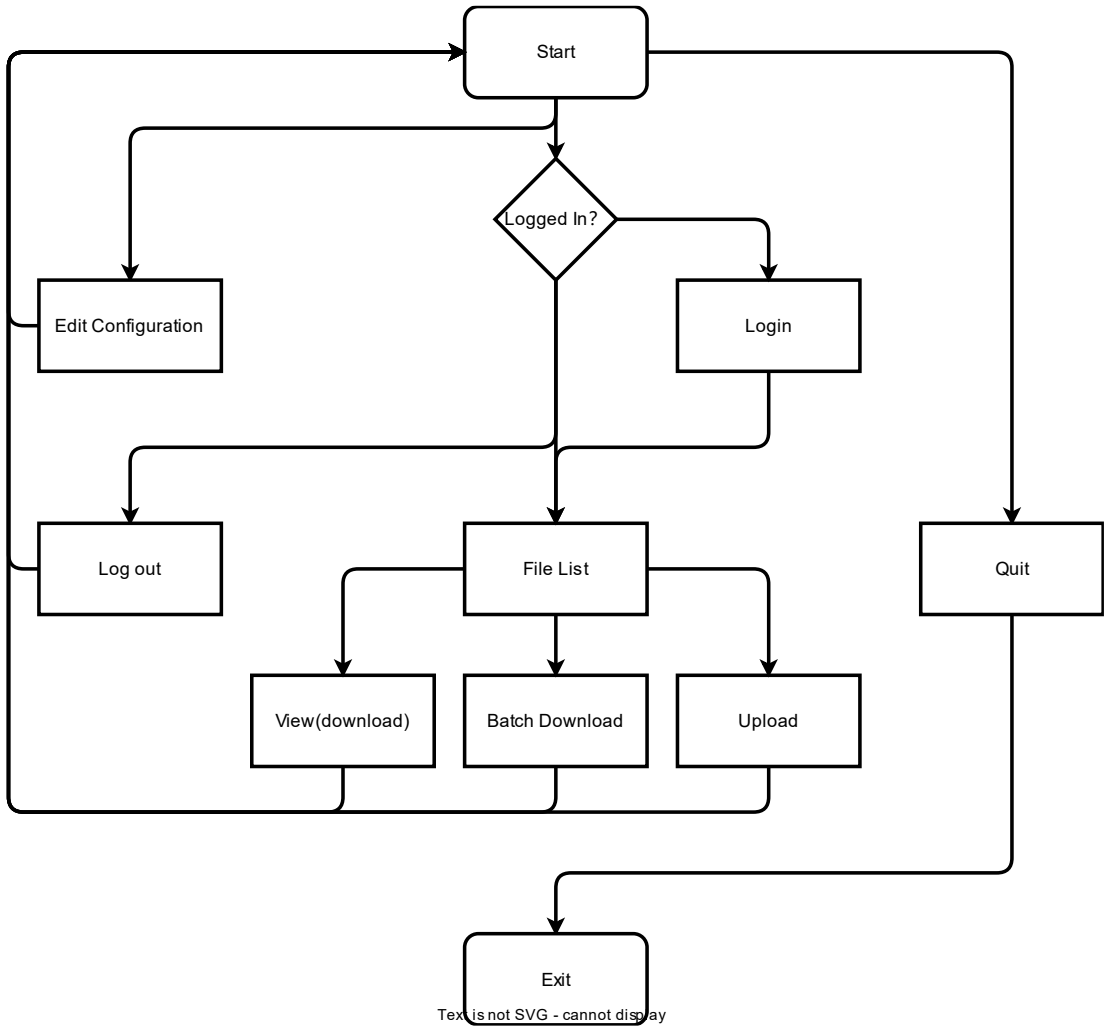
1. 用户启动客户端。
2. 用户输入服务器的 URL 以及身份验证信息。
3. 客户端携带身份验证信息，向服务器发送用于获取可访问文件列表的 GET 请求。
4. 服务器返回文件列表。
5. 用户从列表中选择一个文件进行查看。
6. 客户端向服务器发送用于获取所选文件的 GET 请求。
7. 服务器返回该文件的内容。
8. 客户端将文件内容展示给用户。
9. 用户完成浏览并关闭客户端。

备选流程：

- 如果服务器返回 3xx 状态码，客户端将重定向至新的 URL。
- 如果服务器返回 4xx 或 5xx 状态码，客户端将向用户显示预定义的错误信息。
- 如果服务器返回未知或非预期的状态码，客户端将向用户显示通用错误信息。

更多用例请见源码中的敏捷开发日志 report/client_agile_development.md。

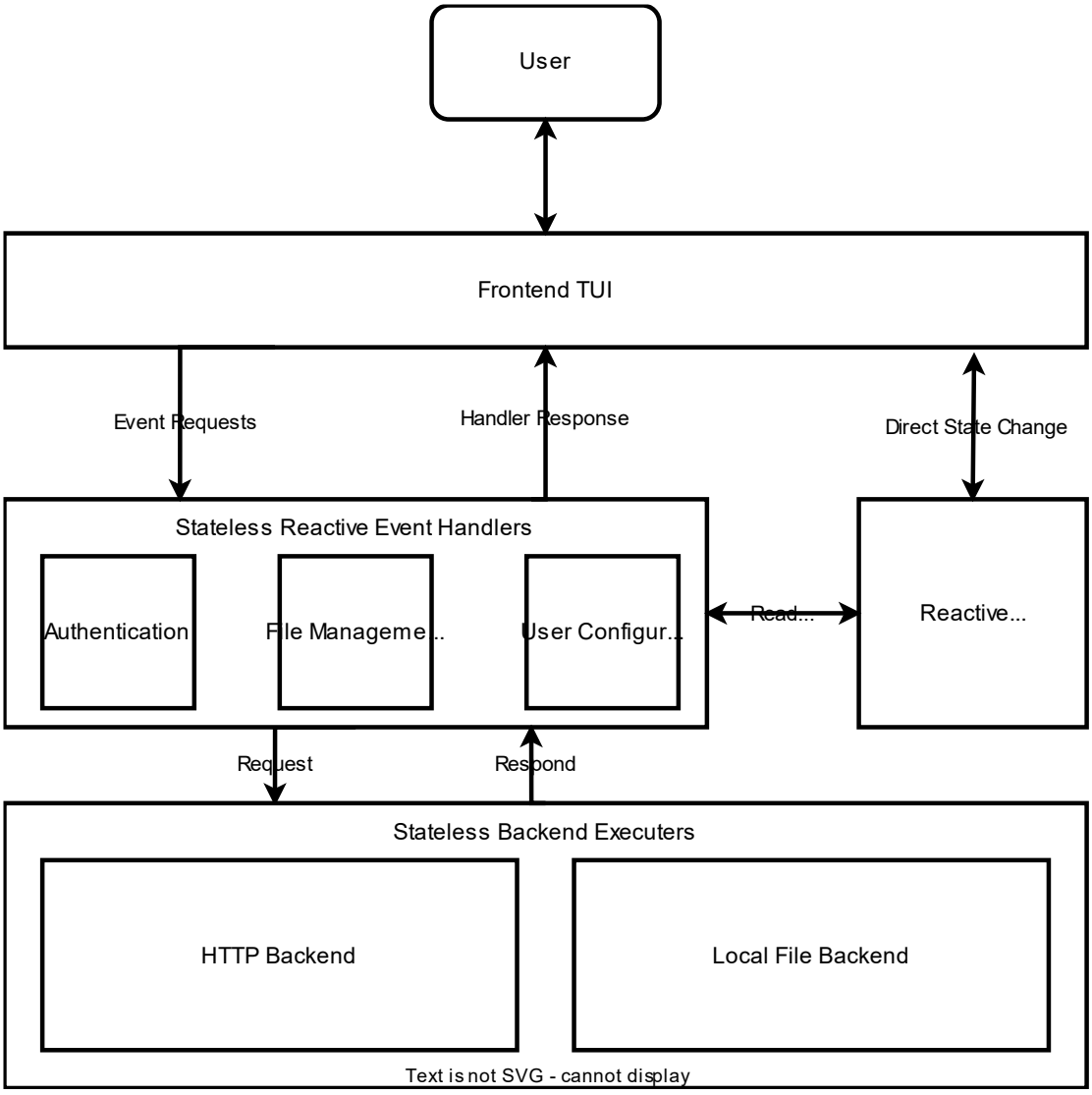
从用数个例中我们总结出用户在不同业务需求之间的专业图。



可以看到，这些业务之间存在着不同的依赖关系。比如，用户需要先登陆才能访问服务器文件系统。但是用户可以随时退出。同时，有些业务之间的发生顺序是无法确定的。例如，用户可能先调整设置，再操作文件，在更换服务器登陆。或者先登录再调整设置。

对于这种实时性高且灵活的用户需求，一个契合要求的程序架构是使用***事件驱动***的处理方式。具体来说，用户（PUBLISHER）可能再任意时刻发送任何请求，这些请求会被识别然后发送到一个对应的响应者（HANDLER）中。这样，一个统一的前端监听事件，再由不同的后端各司，方能灵活响应需求。其职事件驱动的一个难点是处理响应状态（REACTIVE STATE）。比如，用户先调用了一个登陆服务，这个服务产生了一个 SESSION COOKIE。现在用户想调用文件下载服务，此时文件服务的响应者就需要上一个响应者的 COOKIE 来进行服务器授权——这些状态数据的同步对执行业务至关重要。

在我们的实验中，选取了一个轻量且现代化的 TUI 框架 TEXTUAL 来辅助客户端设计（不包含任何网络层工具包，符合实验要求），其提供了成熟的响应状态处理机制。基于 TEXTUAL，我们的整体架构设计如下：



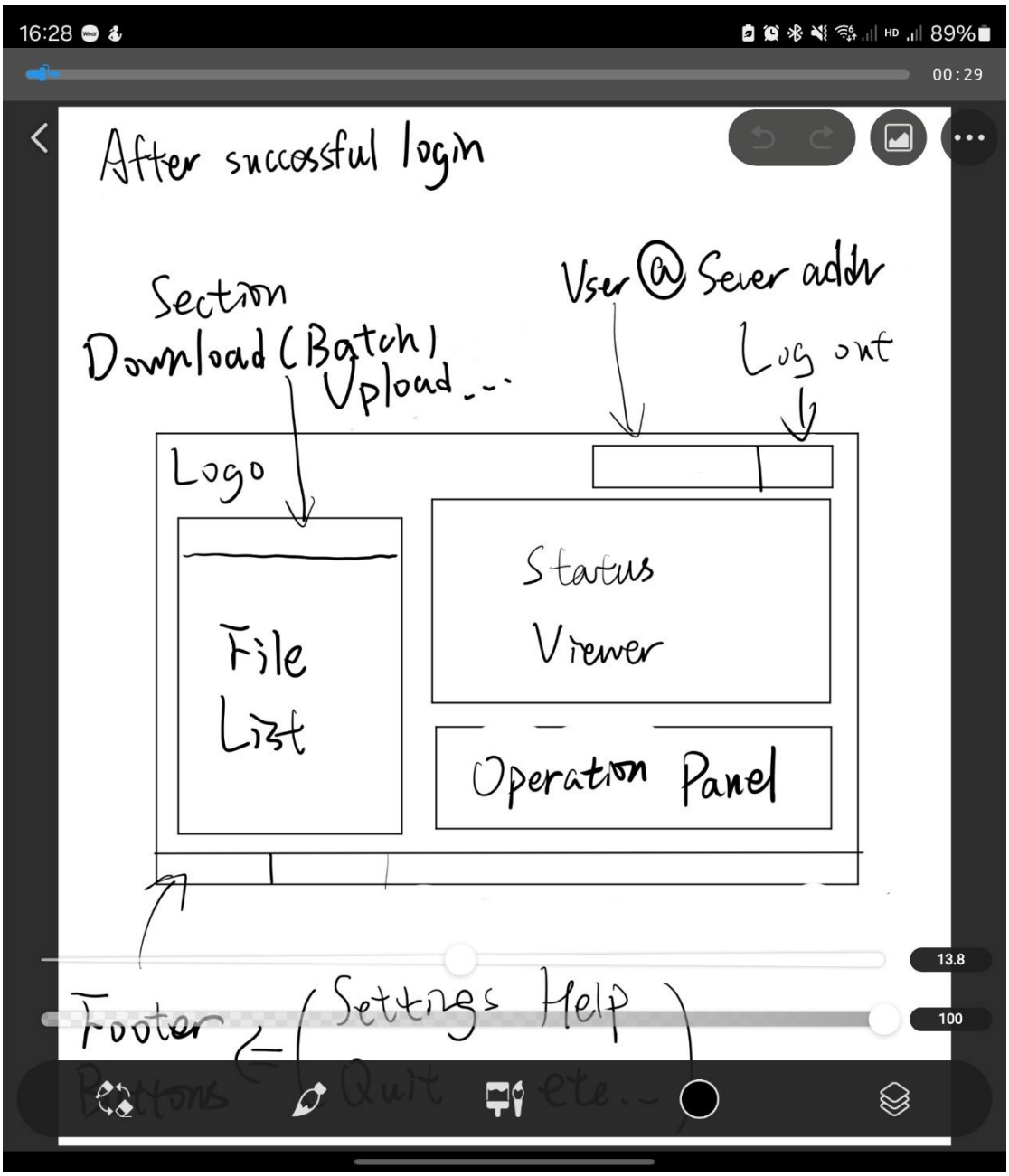
可见其中前端的需求监听由 Textual 提供。我们把用户的业务需求解耦成了三个基本模式，一个是认证，一个是文件管理，一个是软件设置管理——显然，一个用户不可能一边登陆一边下载文件，也不可能在文件传输到一半的时候调成传输设置或更换服务器。这三个基本模式对应了三个响应组件，如图所示。使用 Textual 提供的响应状态控制工具可以有效地控制响应状态的同步，比如 Session Token，服务器地址等。

正如实验要求，这三个组件必然需要与服务器通信（认证、传输文件）和在本地进行文件操作（保存、读取）。我们将其抽象成了两个统一的后端：HTTP 后端和本地文件服务后端。其中 HTTP 后端根据实验要求使用 Socket 实现且不使用高级 HTTP 库，这是本次实验的核心。

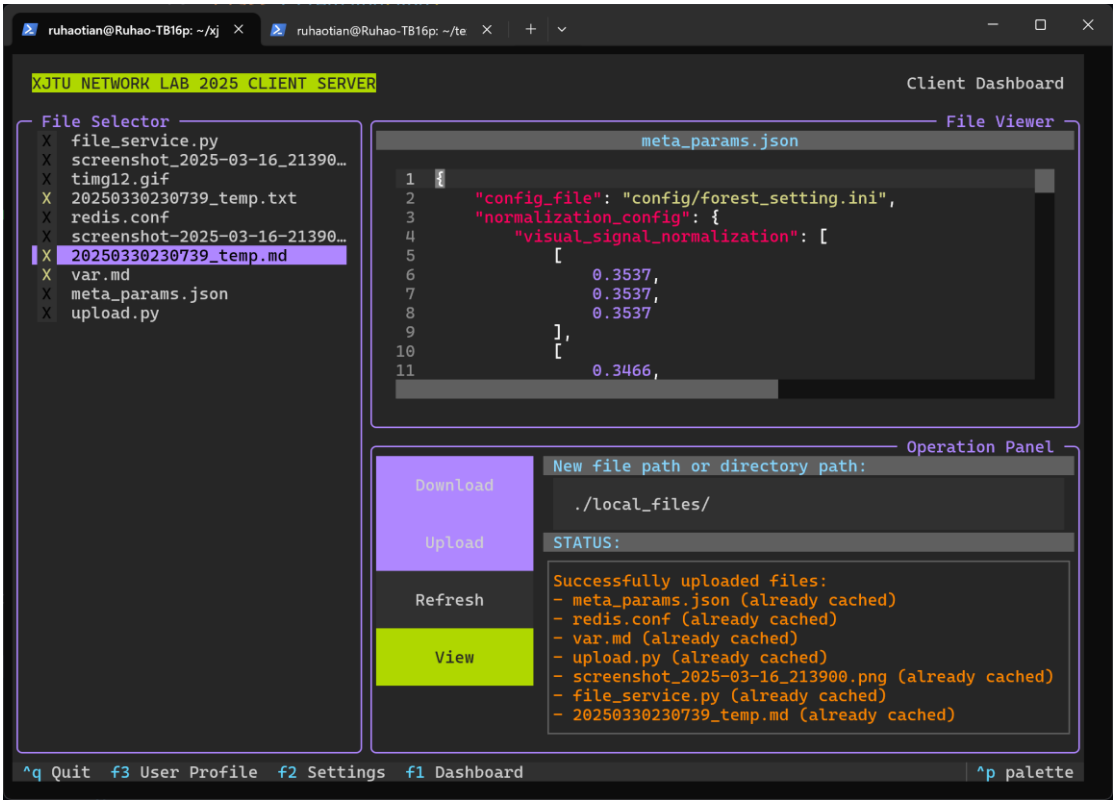
接下来的报告中将根据该架构图展示不同组件的实现。

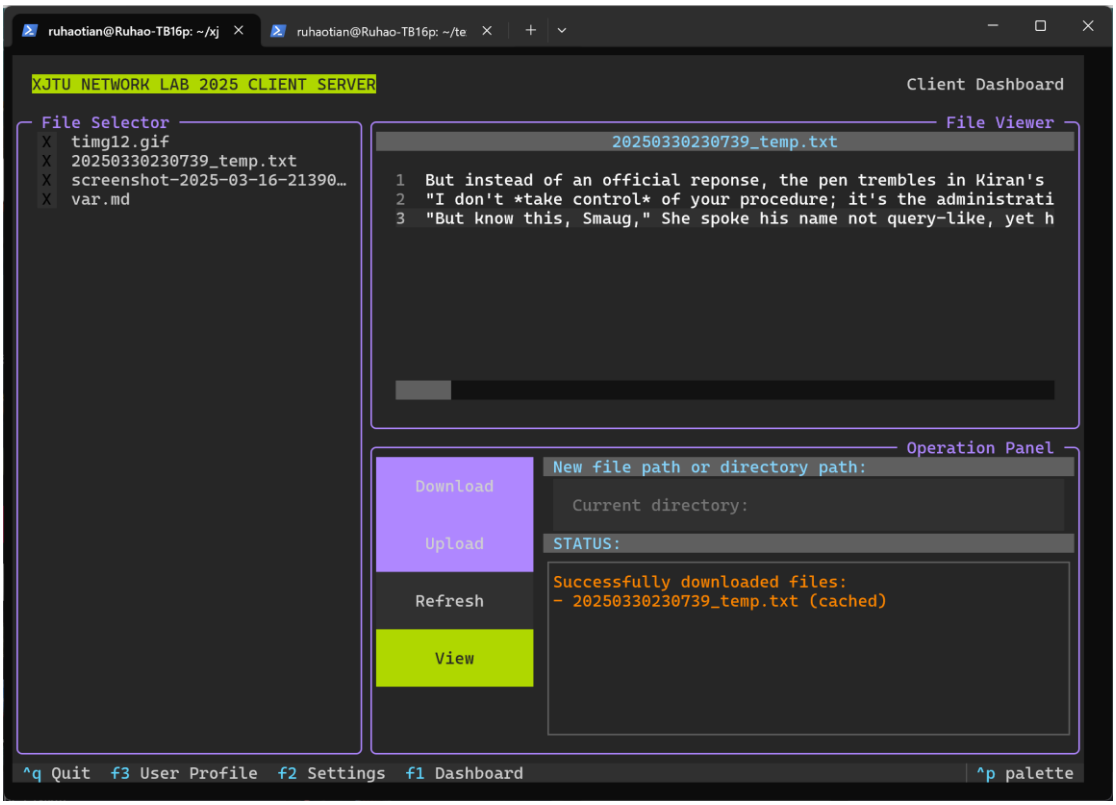
4.2.2 前端 UI 设计

UI 草稿：



实现截图，实际实现有一定差别：





4.2.3 基于 Socket 的 HTTP 通信客户端

HTTP 客户端通过一个统一的端口与上层沟通，这个端口是根据实验要求设计的，主要分为一下部分：

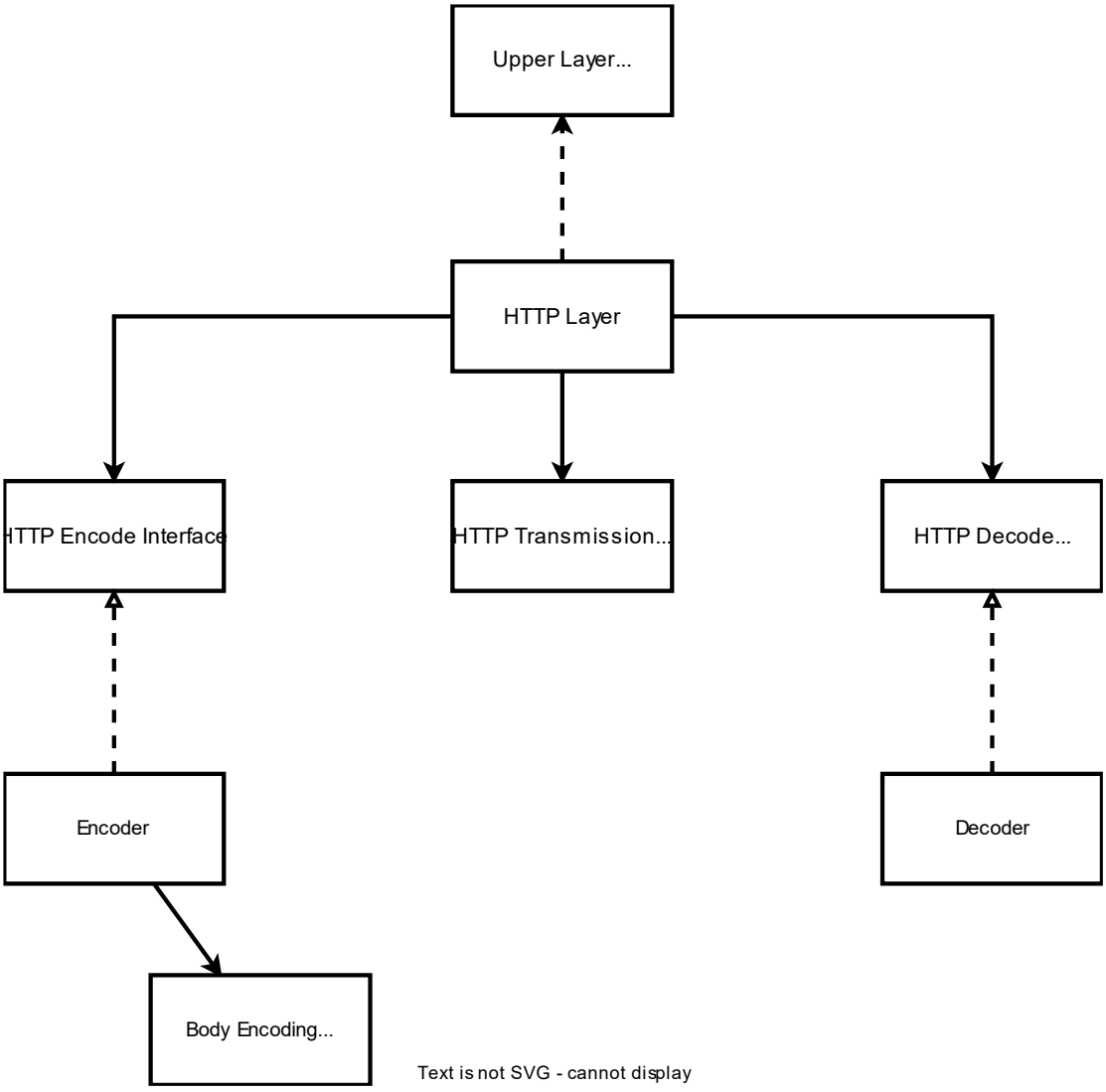
- 发送 HTTP 所必须的地址信息等。
- HTTP 头部选项设置，其中包括实验要求的 User Agent、使用持久连接、使用 Cookie 等。
- 传输选项，包括重定向策略和超时时间等。
- 载荷选项，包括实验要求的分块传输和 gzip 压缩等。

该接口的数据类定义如下：

```
@dataclass
class HTTPLayerInterfaceRequest:
    """A unified Interface for passing a request to the HTTP client.

    Mainly include:
    - request content from the upper layer handlers, i.e.: authenticati
on form
    - global configurations previously defined by the user, i.e.: timeo
ut
    """

    # request model
    url: str
    method: str
    version: str = "HTTP/1.1"

    # server connections
    server_connection: HTTPServerAddress = None

    # request options
    connection_keep_alive: bool = True
    cookie: dict = None
    user_agent: str = None
    accept: str = None
    accept_encoding: str = None

    # transmission options
    timeout: int = 10
    max_retries: int = 3
    allow_redirects: bool = False
    max_redirects: int = 5
    maintain_session_during_redirects: bool = False

    # payload options
    content_length_before_encoding: int = None
    content_encoding: HTTPContentEncoding = None
    transfer_encoding: HTTPTransferEncoding = None
```

```
    transfer_encoding_chunk_size: int = 1024
    payload_type: HTTPPayloadType = None
    payload_bytes: bytes = None
```

尽管选项繁多，在接收到一个请求之后，一个 HTTP 客户端的工作本质上仍然是收发 HTTP 报文。对于一个 HTTP1.1 请求，生成报文、发送报文、接收并解码报文在执行上是独立的，因此将其分为三个功能模块。从而 HTTP 客户端的整体结构如下：



反映到代码上，处理单一报文的顶层函数如下：

```
    def handle_single_request(self, layer_request_interface: HTTPLayerI
nterfaceRequest) -> HTTPLayerInterfaceResponse:
        """Handle a single HTTP request and return the response without
 performing redirection."""
        # Encode the request
        try:
            request_interface = HTTPLayerEncodingModuleInterface(
                url=layer_request_interface.url,
```

```
                method=layer_request_interface.method,
                host=layer_request_interface.server_connection.host_ip,
                version=layer_request_interface.version,
                connection_keep_alive=layer_request_interface.connectio
n_keep_alive,
                cookie=layer_request_interface.cookie,
                user_agent=layer_request_interface.user_agent,
                accept=layer_request_interface.accept,
                accept_encoding=layer_request_interface.accept_encoding
,
                content_encoding=layer_request_interface.content_encodi
ng,
                content_length_before_encoding=layer_request_interface.
content_length_before_encoding,
                transfer_encoding=layer_request_interface.transfer_enco
ding,
                transfer_encoding_chunk_size=layer_request_interface.tr
ansfer_encoding_chunk_size,
                payload_type=layer_request_interface.payload_type,
                payload_bytes=layer_request_interface.payload_bytes,
            )
            encoded_request = self._encode_request(request_interface)
        except ValueError as e:
            print(f"Error encoding request: {e}")
            return HTTPLayerInterfaceResponse(
                http_response=None,
                vaild_response=False,
                error_message="Error encoding request:" + str(e)
            )
        # Create a transmission interface and send the request
        try:
            transmission_interface = HTTPLayerTransmissionModuleInterfa
ce(
                encoded_request=encoded_request,
                timeout=layer_request_interface.timeout,
                max_retries=layer_request_interface.max_retries,
                server=layer_request_interface.server_connection,
            )
            response = self._transmit_request(transmission_interface)
        except TimeoutError as e:
            print(f"Error sending request: {e}")
            return HTTPLayerInterfaceResponse(
                http_response=None,
                vaild_response=False,
```

```
                error_message="Error sending request: " + str(e)
            )
        except Exception as e:
            print(f"Unexpected error: {e}")
            return HTTPLayerInterfaceResponse(
                http_response=None,
                vaild_response=False,
                error_message="Unexpected error: " + str(e)
            )

        # Decode the response
        try:
            decoded_response = self._decode_response(HTTPLayerDecodingM
oduleInterface(
                response_raw_data=response,))
        except Exception as e:
            print(f"Error decoding response: {e}")
            return HTTPLayerInterfaceResponse(
                http_response=None,
                vaild_response=False,
                error_message="Error decoding response: " + str(e)
            )
        return HTTPLayerInterfaceResponse(
            http_response=decoded_response,
            vaild_response=True,
            error_message=None
        )
```

上方的语句大部分都是构造接口数据模型和处理错误，其重点使用高亮标出，可见处理一个报文的核心逻辑无非是先生成报文、传输然后解码返回的报文。

之所以强调处理一个报文，是考虑到重定向请求，所以实际上该函数还会被一个更顶层的函数封装，详见后文。

4.2.3.1 单个报文的生成过程

仅介绍进行生成过程中与实验要求相关的重点细节。

首先是 URL 编码的处理，完成替换不安全字符的要求。

```
def parse_http_url(url: str) -> str:
    """Parse the HTTP URL and return the server address."""
    safe_or_reserved_characters = re.compile(r'[a-zA-Z0-
9\-._~:/?#\[\]@!$&\'()*+,;=]')

    percent_encoded = re.compile(r'%[0-9A-Fa-f]{2}')

    already_encoded = False
```

```
    has_unsafe_characters = False

    # start parsing
    index = 0
    result = ""
    while index < len(url):
        if percent_encoded.match(url[index:index+3]):
            already_encoded = True
            result += url[index:index+3]
            index += 3
        elif safe_or_reserved_characters.match(url[index]):
            result += url[index]
            index += 1
        else:
            has_unsafe_characters = True
            unsafe_bytes = url[index].encode('utf-8')
            for i in unsafe_bytes:
                result += f"%{i:02X}"
            index += 1
    if has_unsafe_characters and already_encoded:
        raise ValueError("URL contains unsafe characters and already en
coded characters, please check the URL")
    return result
```

其逻辑如下：

1. 定义两个正则表达式，一个检查当前字符是否符合 HTTP 规范，一个检查当前三个字符是否能构成一个 URL 编码
2. 开始逐个匹配 URL 中的每个字符
3. 如果不符合规范，那么就将其进行十六进制编码
4. 在 3 发现不安全字符的基础上，如果发现又有已编码的字符，那么报错。
5. 如果没有报错，就返回编码后的字符。

接下来生成请求头，使用一系列判断语句如下：

```
        if encoding_interface.connection_keep_alive:
            headers += "Connection: keep-alive\r\n"
        else:
            headers += "Connection: close\r\n"
```

之后便可以开始处理负载。在我们的客户端设计中，上层负责将文件打包成二进制并提供二进制流长度，由 HTTP 客户端进行编码设置。

其过程用自然语言描述如下：

1. 按接口要求进行 content-encoding
2. 更新 content-length（这是因为 transfer-encoding 是可选的）
3. 按接口要求进行 transfer-encoding

4. 更新 content-length
5. 合并请求头，打包二进制串，提交
6. 如果 1-5 发生错误，返回错误进行处理。

对于 content-encoding，可以直接使用 gzip 等库完成。主要聚焦 transfer-encoding

```
        # apply chunked transfer encoding
        chunked_data = b''
        data_length = len(data)
        offset = 0
        while offset < data_length:
            chunk_size = min(max_chunk_size, data_length - offset)
            chunk = data[offset:offset + chunk_size]
            chunked_data += f"{chunk_size:x}\r\n".encode() + chunk + b"
\r\n"
            offset += chunk_size
```

核心逻辑是对二进制流进行分块，然后按照分块长度添加分块头部说明。当然头部说明也会被转换成二进制添加到负载中。

至此，HTTP 请求的二进制位流生成完成。

4.2.3.2 传输并接收 HTTP 报文

传输所需要实现的功能是 Socket TCP 连接的复用。实现方案是在 HTTP 客户端中维护一个 Socket（本来计划实现连接池，但是没有时间了）。如果服务器返回持久连接，就更新 Socket。每次传输前检查 Socket 可用性，如果可用，按照接口设定决定是否复用 Socket。

```
    def __init__(self):
        # support single connection keep-alive for now
        # TODO: connection pool
        self.persistent_socket = None
        self.current_server: HTTPServerAddress = None
```

为了识别 Socket 属于哪个服务器，还要维护服务器的相关信息。用于检查 Socket 连接可用性的函数如下：

```
    def _check_persistent_socket(self, transmission_interface: HTTPLaye
rTransmissionModuleInterface) -> bool:
        """Check if the persistent socket is still valid."""
        if self.persistent_socket is None:
            return False
        elif self.current_server != transmission_interface.server:
            return False
        try:
            self.persistent_socket.getpeername()
            return True
        except (socket.error, AttributeError):
```

```
            return False
```

首先检查服务器信息是否匹配，然后使用 Socket 发送一条嗅探语句，试探 Socket 是否还维持着连接。

在保证 Socket 连接之后，第二个核心功能是高效的报文接收。Socket 有一个显著的特点：***不负责传输结束控制***。并且使用 Socket 接收时需要指定接收缓冲区大小，如果缓冲区没有充满并且没有超时，那么就会一直等待。

这带来了一个致命的问题：在不知道服务器返回长度的情况下，***如果最后一个缓冲区的长度不能匹配最后一段返回报文，那么每个报文的延迟都至少需要等待一个超时时间***。这对频繁的报文交换和高响应性要求的前端是不可接受的。

解决方法是使用***非阻塞方式***接收 Socket 报文。其基本工作原理如下：

1. Socket 一直工作在非阻塞状态下，也就是一旦受到数据立即返回。理想情况下这会形成一个不断改变的字符流，存放在一个异步缓冲区中。
2. 使用一个轮询语句不断检查这个字符流，查看有没有关键信息（如 content-length 或者结束的回车符号），一旦捕获到，确认报文结束就变得非常简单。
3. 每次轮询之间有一个极短的时间间隔，我们设置为 0.1 秒，来让 Socket 向缓冲区中写入数据。
4. 每次轮询后都会从总超时时间中扣除轮询所用时间

该方法的程序实现如下，注释经过重点标注：

```
# Receive the response
        sock.setblocking(False)  # 非阻塞模式
        data = b''
        header_complete = False
        content_length = 0
        body_start_pos = 0
        # 设置合理的总超时时间
        total_timeout = transmission_interface.timeout
        start_time = time.time()
        while time.time() - start_time < total_timeout:
            try:
                # 使用很短的轮询间隔，不会显著延迟处理
                ready = select.select([sock], [], [], 0.1)  # 100ms 超时
                if ready[0]:
                    chunk = sock.recv(4096)
                    if not chunk:  # 连接关闭
                        # 如果已经接收到完整头部的 GET 请求，可以返回了
                        if header_complete and b'GET' in data[:data.fin
d(b'\r\n')]:
                            return data
                        break
```

```
                    data += chunk
                    # 如果还没解析头部，检查是否已经接收到完整头部
                    if not header_complete and b'\r\n\r\n' in data:
                        header_complete = True
                        body_start_pos = data.find(b'\r\n\r\n') + 4
                        # 解析 Content-Length
                        for line in data[:body_start_pos].split(b'\r\n'
):
                            if line.lower().startswith(b'content-
length:'):
                                content_length = int(line.split(b':', 1
)[1].strip())
                                break

                        # 如果是 GET 请求且没有 Content-Length，可能已经完成
                        if b'GET' in data[:data.find(b'\r\n')] and cont
ent_length == 0:
                            return data
                    # 检查是否已经接收到全部数据
                    if header_complete and len(data) - body_start_pos >
= content_length:
                        return data
                # 如果没有更多数据但头部已完成，可以检查请求类型
                elif header_complete:
                    if b'GET' in data[:data.find(b'\r\n')] or len(data)
 - body_start_pos >=     content_length:
                        return data
            except BlockingIOError:
                # 非阻塞模式下没有数据可读会抛出异常
                pass
        # 超时处理
        raise TimeoutError("HTTP request reception timed out")
```

测试表明该机制能很好的应对频繁的报文传输。

4.2.3.3 解码 HTTP 报文

解码 HTTP 报文的过程基本上是编码 HTTP 报文的逆向过程：

1. 首先提取响应头和响应字段
2. 根据响应头解码负载的 Transfer Encoding
3. 根据响应头解码负载的 Content Encoding

再此不过多赘述。

4.2.3.4 重定向处理

按照实验要求和客户端设计，HTTP 层应该能够将重定向完成的页面交给上层，而不是让上层协助重定向处理。

一个简单的逻辑是跟着重定向地址不断循环请求，直到不再得到 302 状态码为止。但是其中有几个问题值得注意：

- 对于输入密码错误等情况，服务器通常会重定向回登陆界面，这就可能导致无限重定向发生。
- 在重定向期间，可能需要维持 Session Cookie，但是服务器同样可能下发新的 Session Cookie。

为此我们引入了几种控制选项：

- 引入允许重定向选项和最大重定向次数选项。
- 引入维持 Cookie 选项，如果开启，将会在重定向时不断更新 Cookie，并最终返回重定向过程中的最后一个 Cookie。

例如，考虑一个 Cookie 失效的情况。原有 Cookie 失效后将会被重定向到登陆界面。此时需要通过登陆服务来获取新的 Cookie。因此在登陆服务中维持 Cookie 是被开启的。综合以上考量后实现的其核心代码如下：

```
# start redirection loop
            redirect_count = 0
            while redirect_count < max_redirects:
                # check if redirection is needed
                if response.http_response.location:
                    # get the new URL
                    new_url = response.http_response.location
                    print(f"Redirecting to: {new_url}")
                    # create a new request interface
                    layer_request_interface.url = new_url

                    # if maintain_session_during redirects is enabled,
copy the session cookies
                    if maintain_session_during_redirects:
                        # if the response has set-
cookie header, update the cookies
                        if response.http_response.set_cookie != None:
                            print(f"Updating cookies: {response.http_re
sponse.set_cookie}")
                            last_cookie = response.http_response.set_co
okie
                            layer_request_interface.cookie = last_cooki
e
                    # handle the new request
```

```
                    response = self.handle_single_request(layer_request
_interface)
                    # check if redirection is needed
                    if not response.vaild_response:
                        response.error_message = f"Error during redirec
tion: {response.error_message}"
                        return response
                    redirect_count += 1
                else:
                    # this guarantees that the response is 1. valid and
 2. not a redirection
                    # but before returning, apply the last cookie
                    if (not response.http_response.set_cookie) and (las
t_cookie is not None) and (layer_request_interface.maintain_session_dur
ing_redirects):
                        print(f"Applying last cookie: {last_cookie}")
                        response.http_response.set_cookie = last_cookie

                    # return the response
                    return response
            # if max redirects reached, return the response
            response.vaild_response = False
            response.error_message = f"Max redirect count reached: {max
_redirects}"
            return response
```

4.2.4 响应组件设计

根据软件架构图，响应组件介于顶层用户界面和底层 HTTP 后端之间，提供一系列意义明确的方法来快速响应需求。

4.2.4.1 响应状态设计

正如上文所述，响应组件之间的信息需要通过响应状态进行同步。例如当认证服务产生 Session Cookie 后，将其存放在响应状态中，待文件服务需要访问服务器时，就使用尚未失效的 Cookie 进行认证。

我们定义的响应状态主要有两个：Session 状态（包括当前服务器、用户名和 Cookie 等）和 Setting 状态（软件设置，包括默认是否使用长连接，重试和最大重定向次数等）。

Session 状态定义为如下：

```python
@dataclass
class Session:
    """User session model."""
    session_token: str = None
    session_server_info: HTTPServerAddress = None
    # user_name: str
@dataclass
class HTTPServerAddress:
    """HTTP server address model."""
    host_ip: str
    port: int = 80
```

Setting 状态定义如下：

```python
@dataclass
class Setting:
    """Setting model for client configuration."""
    http_request_template: http_model.HTTPLayerInterfaceRequest = None

    auth_service_url: str = "/login"
    file_service_url: str = "/file_service"

    local_file_dir: str = "./local_files/"
@dataclass
class HTTPLayerInterfaceRequest:
    """A unified Interface for passing a request to the HTTP client.

    Mainly include:
    - request content from the upper layer handlers, i.e.: authenticati
on form
    - global configurations previously defined by the user, i.e.: timeo
ut
    """

    # request model
    url: str
    method: str
    version: str = "HTTP/1.1"

    # server connections
    server_connection: HTTPServerAddress = None

    # request options
    connection_keep_alive: bool = True
    cookie: dict = None
    user_agent: str = None
```

```
    accept: str = None
    accept_encoding: str = None

    # transmission options
    timeout: int = 10
    max_retries: int = 3
    allow_redirects: bool = False
    max_redirects: int = 5
    maintain_session_during_redirects: bool = False

    # payload options
    content_length_before_encoding: int = None
    content_encoding: HTTPContentEncoding = None
    transfer_encoding: HTTPTransferEncoding = None
    transfer_encoding_chunk_size: int = 1024
    payload_type: HTTPPayloadType = None
    payload_bytes: bytes = None
```

4.2.4.2 认证服务设计

认证服务面上上层提供网站登陆。其接收一个 Credential 数据类（包含服务器、用户名、密码等）和当前 Setting 状态，并返回一个 AuthResult 数据类，其中包含认证是否成功、得到的 Cookie（如果成功）和错误语句等。

```
@dataclass
class Credentials:
    """User credentials model."""
    server_address: str
    username: str
    password: str
@dataclass
class AuthResult:
    """Authentication result model."""
    success: bool
    session_model: Optional[Session] = None
    error_message: Optional[str] = None
```

由于下层 HTTP 后端已经封装了底层传输细节，认证服务只需配置一个 HTTP 传输请求，然后接收结果即可。

```
    async def login(self, credentials: Credentials, setting: Setting =
None) -> AuthResult:
        """Authenticate user with the server."""
        try:

            if setting is None:
```

```
                setting = replace(Setting())
                setting.http_request_template = DEFAULT_HTTP_REQUEST_TE
MPLATE

            server_info = HTTPServerAddress(
                host_ip=credentials.server_address,
                port=80,
            )
            auth_request = replace(setting.http_request_template)
            auth_request.url = setting.auth_service_url
            auth_request.method = "POST"
            auth_request.server_connection = server_info
            auth_request.payload_type = HTTPPayloadType.FORM_URLENCODED
            auth_request.payload_bytes, auth_request.content_length_bef
ore_encoding = encode_auth_form(credentials)
            auth_request.allow_redirects = True
            auth_request.maintain_session_during_redirects = True

            response = self.http_client.handle_request(auth_request)
```

可见，选用了 POST 方法并将认证信息编码成表单，同时打开了必要的重定向功能。

在接收到请求后，服务器还要根据状态码进行错误处理，并生成 AuthResult 数据。此部分实现比较直观，仅展示一个根据状态码判断响应状态的部分。

```
                        # check status code
                        error_message = handle_common_http_error(respon
se.http_response.status_code)
                        # in this specific case, if the auth info is in
valid, the server will always redirect to the login page, until the max
imum number of redirects is reached
                        # TODO: better handle this case
                        if response.http_response.payload_bytes != None
:
                            error_message = "Invalid username or passwo
rd."
                        elif error_message == None:
                            error_message = f"Unknown error occurred, s
tatus code: {response.http_response}"
                        else:
                            pass
```

首先，handle_common_http_error 处理常见状态码，比如 404 或服务器内部错误。其次，如果在重定向后还是没有 Body 数据，并且也不是常见状态码，此时表明不断重定向到登陆界面直到达到了重定向次数上限，意味着用户名或密码错误。

4.2.4.3 文件服务设计

文件服务提供几个不同请求处理：

- 从 Server 获取当前文件列表
- 从 Server 单个或批量下载文件
- 向 Server 单个或批量上传文件

从服务器获取文件的流程和认证服务原理一致：创建一个自定义 HTTP 请求，在负载中加上 API 表单，发送并接收数据，接收后进行错误处理。此处仅展示自定义的 API：

```
@dataclass
class SingleFile:
    """Model for a single file."""
    file_name: str = None
    file_hash: Optional[str] = None
    file_data: Optional[bytes] | Optional[str] = None
@dataclass
class ServerFileList:
    """Model for a list of files on the server."""
    valid_list: bool = False
    file_list: list[SingleFile] = None
    error_message: Optional[str] = None
@dataclass
class FileServerRequestAPI:
    """Model for file server request API. This is the payload for the f
ile server request."""
    request_type: FileServerRequestType = None
    request_download_file_list: Optional[list[SingleFile]] = None
    request_upload_file_list: Optional[list[SingleFile]] = None
@dataclass
class FileServerResponseAPI:
    """Model for file server response API. This is the payload for the
file server response."""
    request_success: bool = False
    request_data: Optional[list[SingleFile]] = None
    error_message: Optional[str] = None
```

文件下载和上传服务的逻辑相对复杂，涉及到缓存机制。其步骤统一如下：

- 首先调用文件服务自身检查服务器/本地的文件列表
- 将要上传/下载的文件列表与服务器/本地的文件列表做差，找到可能作为缓存的文件名。
- 对可能缓存的文件进行 MD5 哈希校验，排除已经缓存的文件，生成最终需要上传/下载的文件列表。

- 发送请求。
- 处理相应并生成结果，添加错误语句等。

此处展示下载服务是如何进行缓存校验的：

```
            # 1. get server file list
            fetch_file_list_request = FetchServerFileInterface(
                current_session=layer_request_interface.current_session
,
                setting=layer_request_interface.setting,
            )
            current_server_file_list = self.fetch_server_file_list(
                fetch_file_list_request
            )
            if not current_server_file_list.valid_list:
                raise RuntimeError(
                    "Error fetching server file list before download: "
                    + current_server_file_list.error_message
                )
            # 2. match download file from server file list
            # if not found, raise error
            file_name_list = layer_request_interface.file_name_list
            request_download_file_info_list: list[SingleFile] = []
            for file_name in file_name_list:
                if not any(
                    file.file_name == file_name
                    for file in current_server_file_list.file_list
                ):
                    raise RuntimeError(f"File '{file_name}' not found o
n server.")
                else:
                    request_download_file_info_list.append(
                        next(
                            file
                            for file in current_server_file_list.file_l
ist
                            if file.file_name == file_name
                        )
                    )
            # 3. cache mechanism: check local file & hash
            # if local file exists and hash matches, skip download
            # if local file exists but hash doesn't match, raise error
            # TODO: more ways to handle file conflict
            actual_download_file_info_list: list[SingleFile] = []
            for file_info in request_download_file_info_list:
```

```
                local_file_path = (
                    layer_request_interface.setting.local_file_dir + fi
le_info.file_name
                )
                if os.path.exists(local_file_path):
                    local_file_hash = self.local_file_backend.get_file_
hash(
                        local_file_path
                    )
                    if local_file_hash == file_info.file_hash:
                        # skip download
                        continue
                    else:
                        raise RuntimeError(
                            f"Local file '{file_info.file_name}' exists
 but has hash mismatch."
                        )
                else:
                    # add to download list
                    actual_download_file_info_list.append(file_info)
            # 4. download files
```

5 测试及结果分析

考虑到项目代码结构复杂，我们引用了 Pytest 测试框架来构建标准化测试。在构建测试时，我们同样将测试按照客户端的架构模块进行分类。

- 对 Textual 前端，其异步特性导致并不支持 Pytest。虽然有专用的测试框架，但是实现复杂，最终我们进行手动测试。
- 对本地文件管理后端，其实现非常简单，因此没有单独设置测试，而是将其集成在上层响应服务中一同测试
- 对于认证服务、HTTP 后端、文件服务，分别撰写测试脚本进行单独测试。
- 测试时使用 YAML 进行数据与测试用例分离。

最终测试系统的文件结构如下。

```
testing
 ├ __pycache__
 ├ test_auth
 │ ├ __pycache__
 │ ├ test_auth_cases.yaml
 │ └ test_auth_service.py
 ├ test_file_service
 │ ├ __pycache__
```

```
│ ├ downloaded_files
│ ├ upload_files
│ ├ test_file_service.py
│ └ test_file_service_cases.yaml
└ test_http
  ├ __pycache__
  ├ __init__.py
  ├ test_http_cases.yml
  └ test_http_service.py
```

得益于 Pytest 灵活的控制能力，我们可以单独选取测试运行并进行抓包分析。

测试的服务器端使用一个简单配置的 Apache2 服务器，配置过程省略，仅满足客户端 API 要求即可。

由于 Textual 库的异步属性，导致无法使用 Pytest 进行直接测试，因此我们手动进行 UI 测试。

5.1 HTTP 后端测试

我们使用 Pytest 设计了 7 个不同的 HTTP 单元测试，每一个都对应不同的功能需求：

测试名称	对应需求
test_http_client_socket_minimal	基本网站访问
test_http_client_socket_form_login	Session Cookie 和网站登陆
test_http_client_socket_form_login_send_encoding	不同的客户端编码方式
test_http_client_socket_minimal_respond_encoding	解码不同的服务器报文
test_http_client_socket_form_login_with_redirection	带有重定向的登陆测试
test_http_client_socket_redirection_respond_infinite	测试无限重定向的自动阻止
test_http_client_socket_parse_url	访问有不安全字符的 URL

运行测试并进行抓包，可见测试全部通过。下面对每个测试结果进行分析。***请注意所有的抓包结果 User Agent 均包含本人英文名。***

```
(temp) ruhaotian@Ruhao-TB16p:~/xjtu_ns_exp_project$ cd client/
(temp) ruhaotian@Ruhao-TB16p:~/xjtu_ns_exp_project/client$ pytest -v ./testing/test_http/test_http_service.py
============================= test session starts ==============================
platform linux -- Python 3.12.4, pytest-8.3.5, pluggy-1.5.0 -- /home/ruhaotian/miniconda3/envs/temp/bin/python
cachedir: .pytest_cache
rootdir: /home/ruhaotian/xjtu_ns_exp_project/client
configfile: pytest.ini
plugins: anyio-4.9.0
collected 7 items

testing/test_http/test_http_service.py::test_http_client_socket_minimal PASSED [ 14%]
testing/test_http/test_http_service.py::test_http_client_socket_form_login PASSED [ 28%]
testing/test_http/test_http_service.py::test_http_client_socket_form_login_send_encoding PASSED [ 42%]
testing/test_http/test_http_service.py::test_http_client_socket_minimal_respond_encoding PASSED [ 57%]
testing/test_http/test_http_service.py::test_http_client_socket_form_login_with_redirection PASSED [ 71%]
testing/test_http/test_http_service.py::test_http_client_socket_form_login_with_redirection_respond_infinite PASSED [ 85%]
testing/test_http/test_http_service.py::test_http_client_socket_parse_url PASSED [100%]

============================== 7 passed in 1.00s ===============================
(temp) ruhaotian@Ruhao-TB16p:~/xjtu_ns_exp_project/client$
```

5.1.1 基本网站访问

测试使用的接口设置如下：

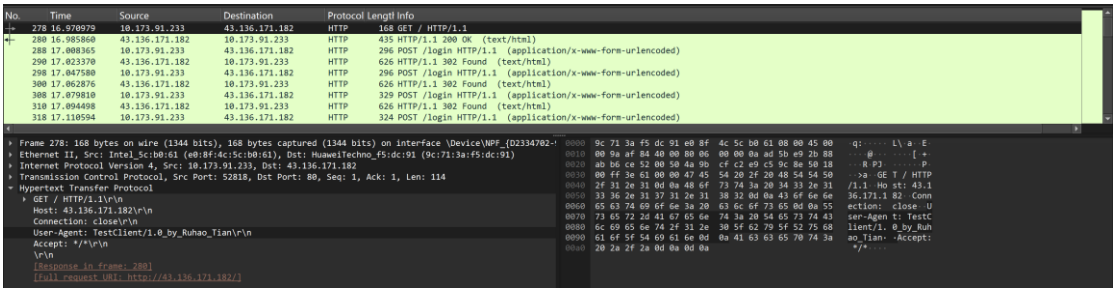
```yaml
test_http_request_minimal:
  url: /
  method: GET
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding: Null
  transfer_encoding_chunk_size: Null
  content_encoding: Null
  payload_type: Null
  payload_data: NULL
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: False
  max_redirects: 0
  maintain_session_during_redirects: False
  expected_response:
    valid_response: true
    status_code: 200
```

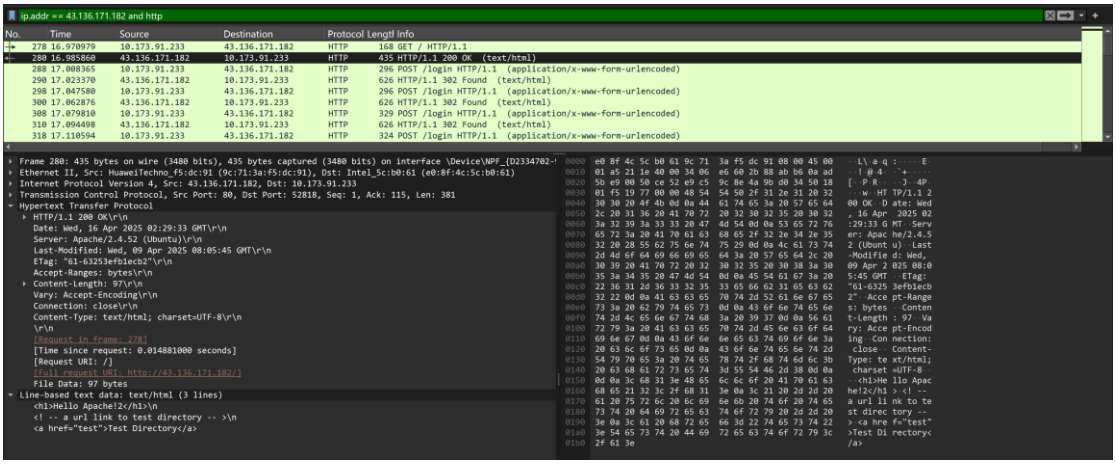
可见，所有的可选字段都被置空。仅测试最基本的访问请求。测试程序先加载数据，发送请求并判断响应是否满足设置中的 expected_response 中设置的条件，比如状态码为 200 等。

```python
def test_http_client_socket_minimal():
    """Test the HttpClientSocket class."""
    # Load test data
    test_data = load_test_data("./testing/test_http/test_http_cases.yml
")
    test_data = test_data["test_http_request_minimal"]
    # Initialize the HttpClientSocket
    client_socket = HttpClientSocket()
    # Test data for HTTP request
    http_request = fetch_http_request_data(test_data)
    # Call the handle_request method and check the response
    response = client_socket.handle_request(http_request)
```

```
    assert isinstance(response, HTTPLayerInterfaceResponse)
    assert response.vaild_response == test_data["expected_response"]["v
alid_response"]
    assert (
        response.http_response.status_code
        == test_data["expected_response"]["status_code"]
    )def test_http_client_socket_minimal():
    """Test the HttpClientSocket class."""
    # Load test data
    test_data = load_test_data("./testing/test_http/test_http_cases.yml
")
    test_data = test_data["test_http_request_minimal"]
    # Initialize the HttpClientSocket
    client_socket = HttpClientSocket()
    # Test data for HTTP request
    http_request = fetch_http_request_data(test_data)
    # Call the handle_request method and check the response
    response = client_socket.handle_request(http_request)
    assert isinstance(response, HTTPLayerInterfaceResponse)
    assert response.vaild_response == test_data["expected_response"]["v
alid_response"]
    assert (
        response.http_response.status_code
        == test_data["expected_response"]["status_code"]
    )
```

截获的请求响应报文如下，可见均符合测试设计。





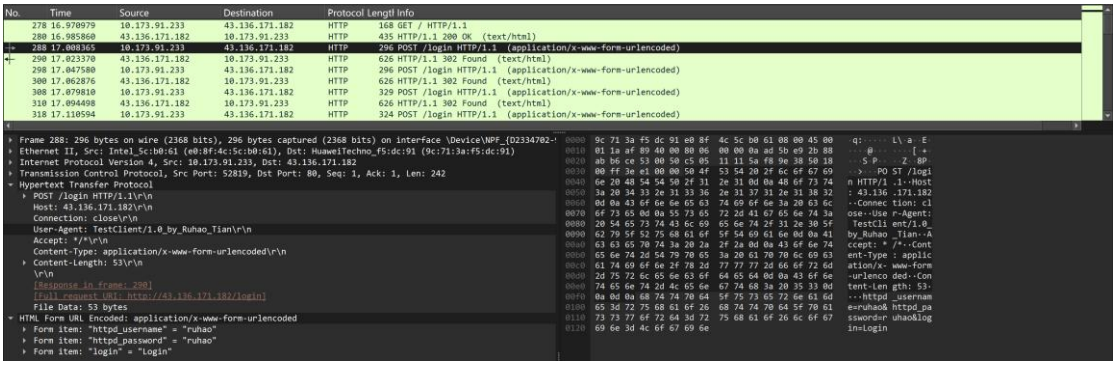
5.1.2 Session Cookie 和网站登陆

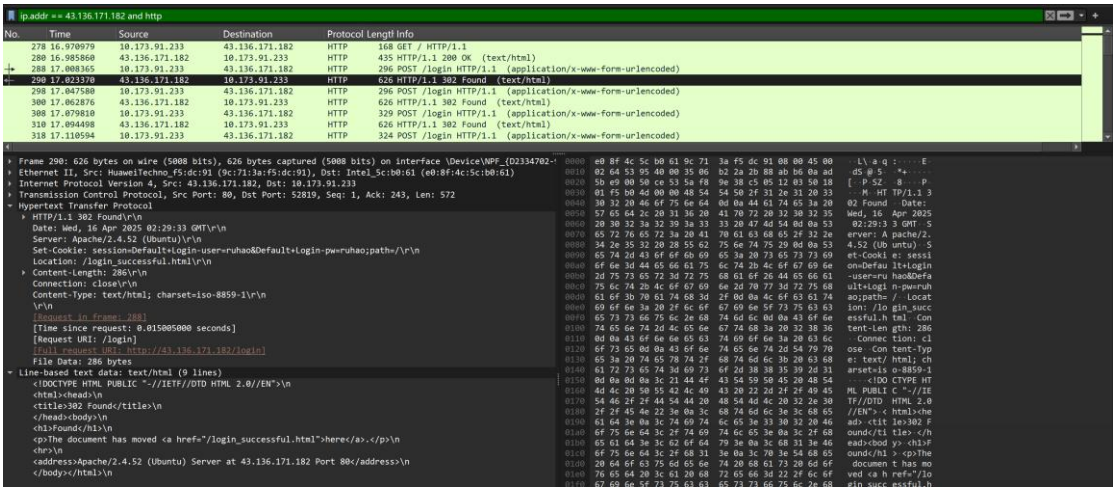
测试配置：

```yaml
test_http_request_form_login:
  url: /login
  method: POST
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding: Null
  transfer_encoding_chunk_size: Null
  content_encoding: Null
  payload_type: "application/x-www-form-urlencoded"
  payload_data:
    httpd_username: *username
    httpd_password: *userpassword
    login: "Login"
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: False
  max_redirects: 0
  maintain_session_during_redirects: False
  expected_response:
    valid_response: false # because redirect is not allowed
    status_code: 302 # redirect to login success page
```

```
    location: "/login_successful.html" # despite the redirect, a login success page is expected
    error_message: "Redirection is needed, but not allowed"
```

请注意此处并没有开启重定向。因此期望状态码 302，并且指向登陆成功界面。





抓包结果与 Pytest 程序判断完全相同。

5.1.3 不同的客户端编码方式

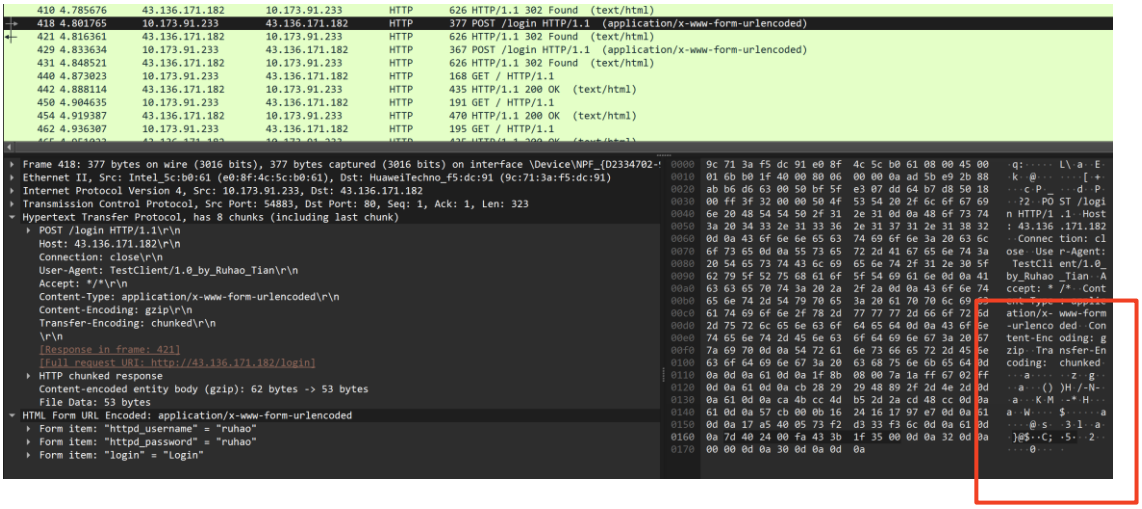
测试配置：

```
test_http_request_form_login_send_encoding:
  url: /login
  method: POST
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding:
    - NULL
    - "chunked"
```

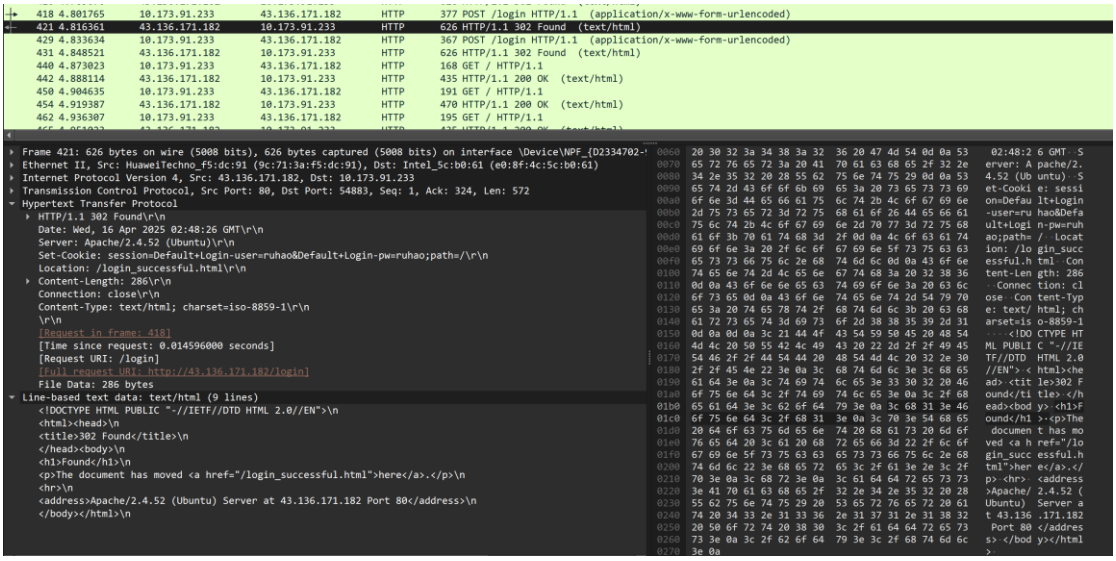
```
    #- "identity"
  transfer_encoding_chunk_size: 10
  content_encoding:
    - NULL
    - "gzip"
    - "identity"
  payload_type: "application/x-www-form-urlencoded"
  payload_data:
    httpd_username: *username
    httpd_password: *userpassword
    login: "Login"
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: False
  max_redirects: 0
  maintain_session_during_redirects: False
  expected_response:
    valid_response: false # because redirect is not allowed
    status_code: 302 # redirect to login success page
    location: "/login_successful.html" # despite the redirect, a login
success page is expected
    error_message: "Redirection is needed, but not allowed"
```

请注意此处 transfer-encoding 和 content-encoding 均为列表。测试程序会***遍历所有的 Encoding 组合并对每一个响应进行判断。***

此处仅选取一个典型抓包结果。

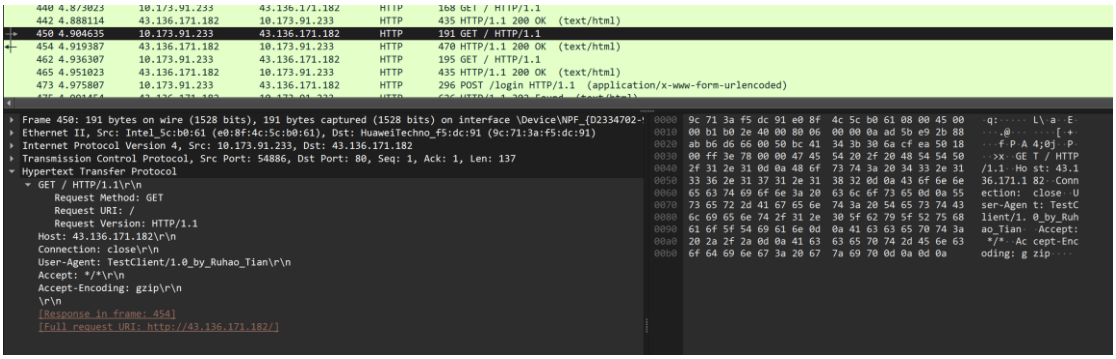


在这个报文中，Content 使用 gzip 进行压缩，压缩后的二进制位流使用 chunk 编码进一步分块。可见服务器响应正常（未开启重定向，所以期望 302，详见测试配置）。

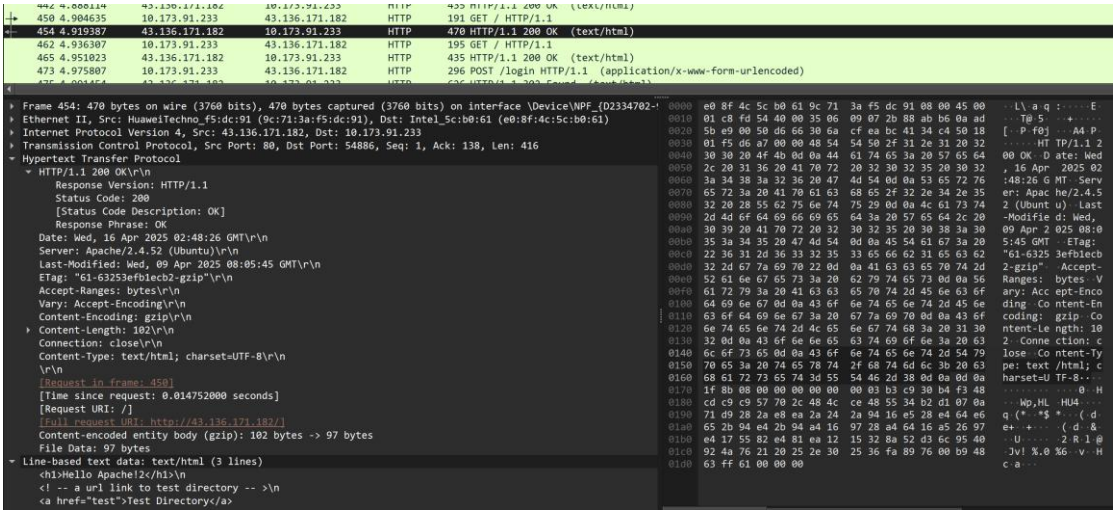


5.1.4 解码不同的服务器报文

与上方测试设计完全一致，此处直接给出一个典型结果。



客户端请求一个 gzip 压缩的报文。



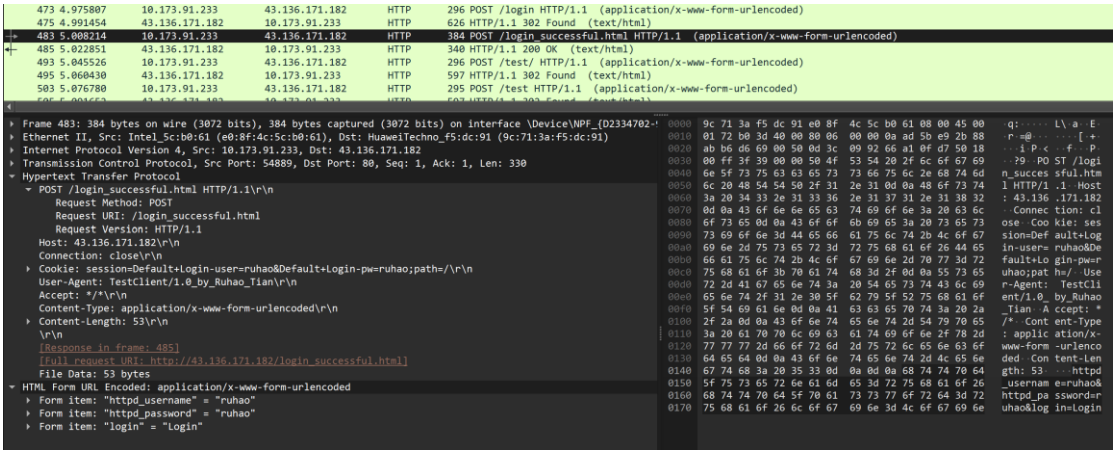
服务端返回报文并且能被正确解码。

5.1.5 带有重定向的登陆测试

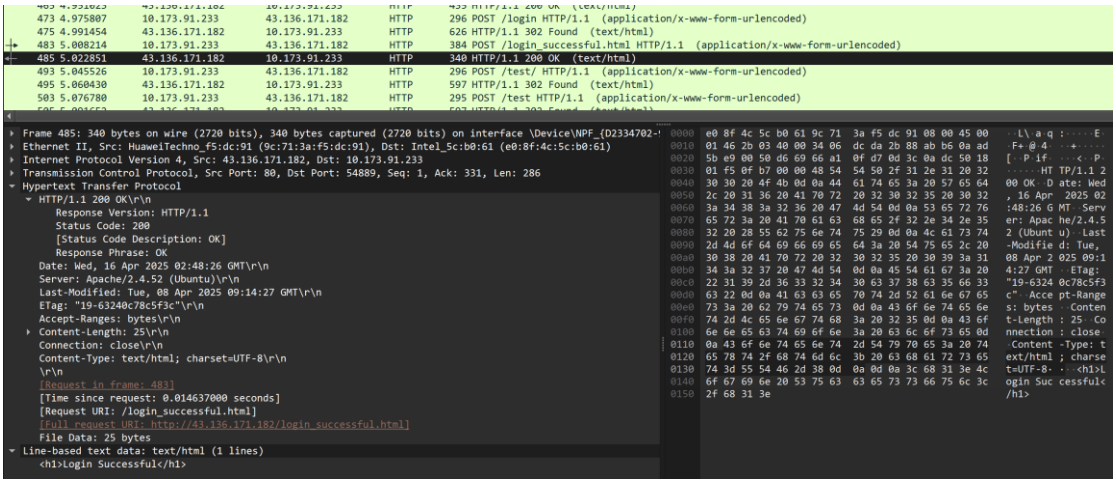
测试配置：

```yaml
test_http_request_form_login_with_redirection:
  url: /login
  method: POST
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding: Null
  transfer_encoding_chunk_size: Null
  content_encoding: Null
  payload_type: "application/x-www-form-urlencoded"
  payload_data:
    httpd_username: *username
    httpd_password: *userpassword
    login: "Login"
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: True
  max_redirects: 10
  maintain_session_during_redirects: True
  expected_response:
    valid_response: True
    status_code: 200 # redirect to login success page
    error_message: Null
```

可见测试重定向选项均被开启了，并且期望得到状态码 200.



抓包发现，在 473 行进行 Login 操作并收到 302 请求（475 行）后，客户端再次请求了/login_successful.html，并且带上了先前的 Cookie（483 行）



服务器也返回了期望的结果。

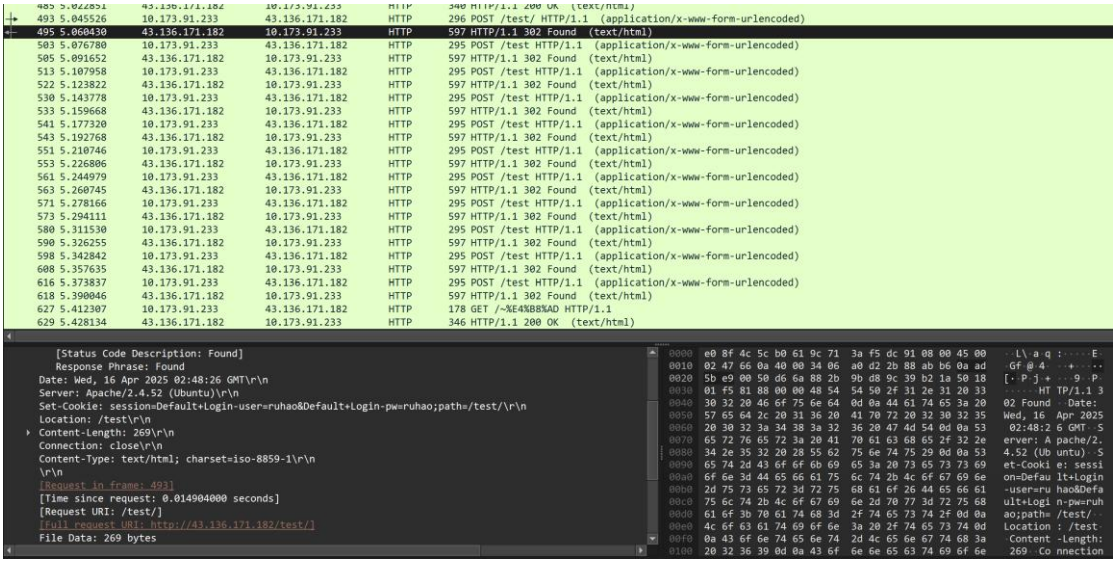
5.1.6 测试无限重定向的自动阻止

测试配置：

```yaml
test_http_request_form_login_with_redirection_infinite:
  url: /test/
  method: POST
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding: Null
  transfer_encoding_chunk_size: Null
  content_encoding: Null
```

```yaml
  payload_type: "application/x-www-form-urlencoded"
  payload_data:
    httpd_username: *username
    httpd_password: *userpassword
    login: "Login"
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: True
  max_redirects: 10
  maintain_session_during_redirects: False
  expected_response:
    valid_response: False
    error_message: "Max redirect count reached: 10"
```

该测试请求的 URL 是经过特别设置的，一旦访问这个 URL 服务器就会返回重定向报文，重定向地址还是该 URL 本身，以此来触发无限重定向。此处将最大重定向次数设置为 10.



可见从首个重定向开始，恰好有 10 个重定向请求。

5.1.7 访问有不安全字符的 URL

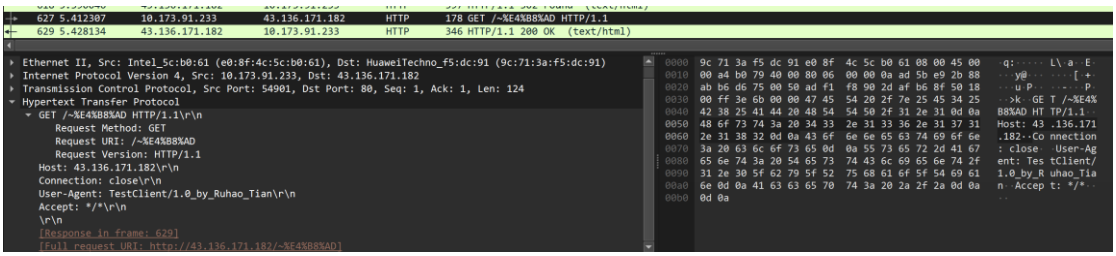
在服务器中设置一个带有不安全符号的 URL，指向一个测试成功界面。

```
<VirtualHost *:80>

    DocumentRoot "${APACHE_SERVER_DIR}/doc_root"

    DirectoryIndex index.html

    Alias /~中 /${APACHE_SERVER_DIR}/doc_root/url_parse_successful.html
```

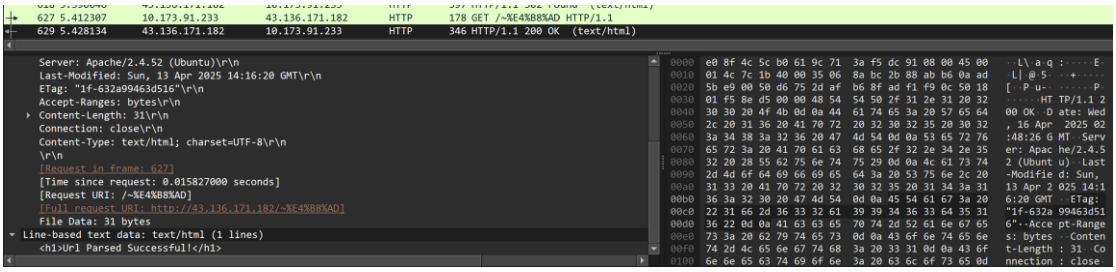
测试配置：

```yaml
test_http_request_parse_url:
  url: /~中
  method: GET
  version: "HTTP/1.1"
  host: *server_host
  connection_keep_alive: false
  cookie: Null
  user_agent: *user_agent
  accept: "*/*"
  accept_encoding: Null
  transfer_encoding: Null
  transfer_encoding_chunk_size: Null
  content_encoding: Null
  payload_type: Null
  payload_data: Null
  server_connection:
    host_ip: *server_ip
    port: 80
  timeout: 5
  max_retries: 3
  allow_redirects: False
  max_redirects: 0
  maintain_session_during_redirects: False
  expected_response:
    valid_response: true
    status_code: 200
    error_message: Null
```

写入不安全的 URL。



抓包可见 URL 中的不安全字符被自动替换。



服务器也返回了正确的结果。

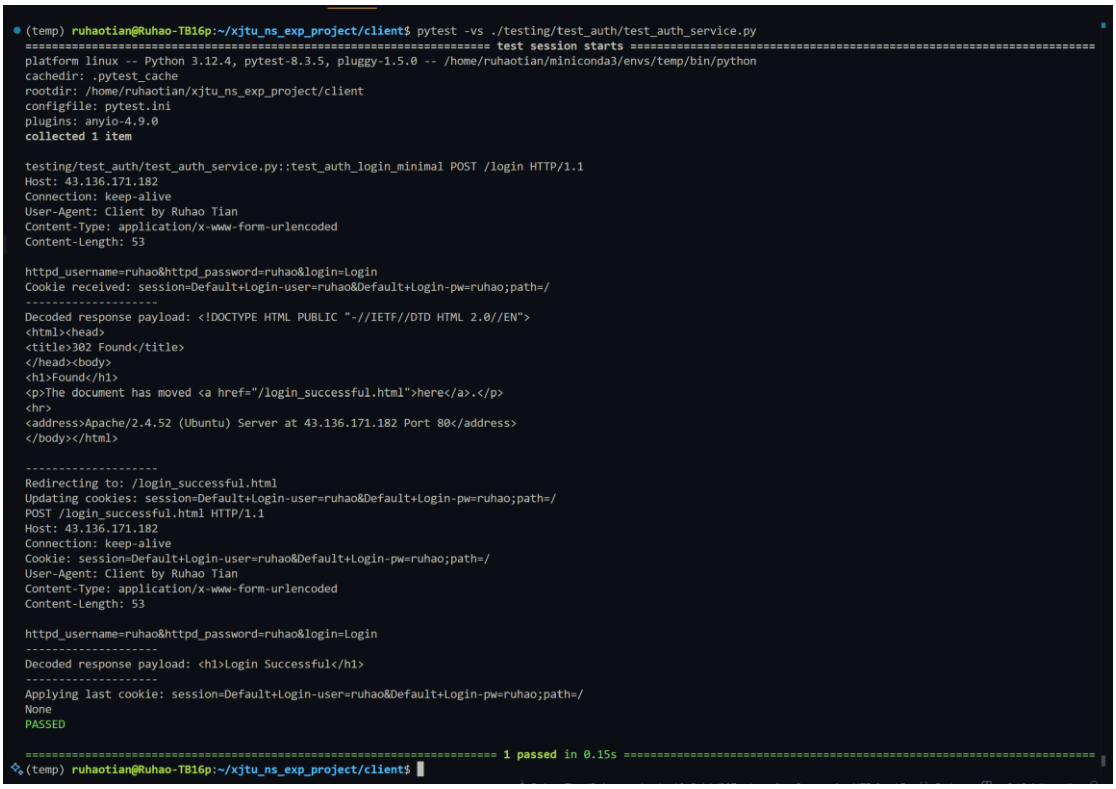
5.2 认证服务测试

5.1 章节已经保证了 HTTP 底层的传输稳定性，因此我们通过 Pytest 输出语句直接观察测试结果，这也是使用层次化设计的一个优势。

由于认证服务只有一个上层接口，我们设计了一个对应的测 test_auth_login_minimal 验证其功能。

```
def test_auth_login_minimal():
    """Test the AuthService login method with minimal data."""

    #load test data
    test_data = load_test_data('./testing/test_auth/test_auth_cases.yaml')
    test_data = test_data['test_auth_login_minimal']

    test_credentials = Credentials(
        server_address=test_data['credentials']['server_address'],
        username=test_data['credentials']['username'],
        password=test_data['credentials']['password']
    )

    # initialize the AuthService
    http_client = HttpClientSocket()
    auth_service = AuthService(http_client)
    response = asyncio.run(auth_service.login(test_credentials))
    print(response.error_message)
    assert response.success == test_data['expected']['success']
    assert response.session_model.session_token != None
```

可见其输入证书后检查 1.回复是否有效 2.是否得到了 Session Cookie。



可见测试通过，并且解码的报文显示底层 HTTP 服务成功处理了重定向请求，并获得了 Session Cookie。请注意 User Agent 是本人的英文名。

5.3 文件服务测试

测试名称	对应需求
test_fetch_file_list	尝试获取服务器文件列表
test_download_all_file	测试下载服务器的所有文件
test_download_all_with_cache	测试下载已缓存的文件
test_upload_file	测试文件上传

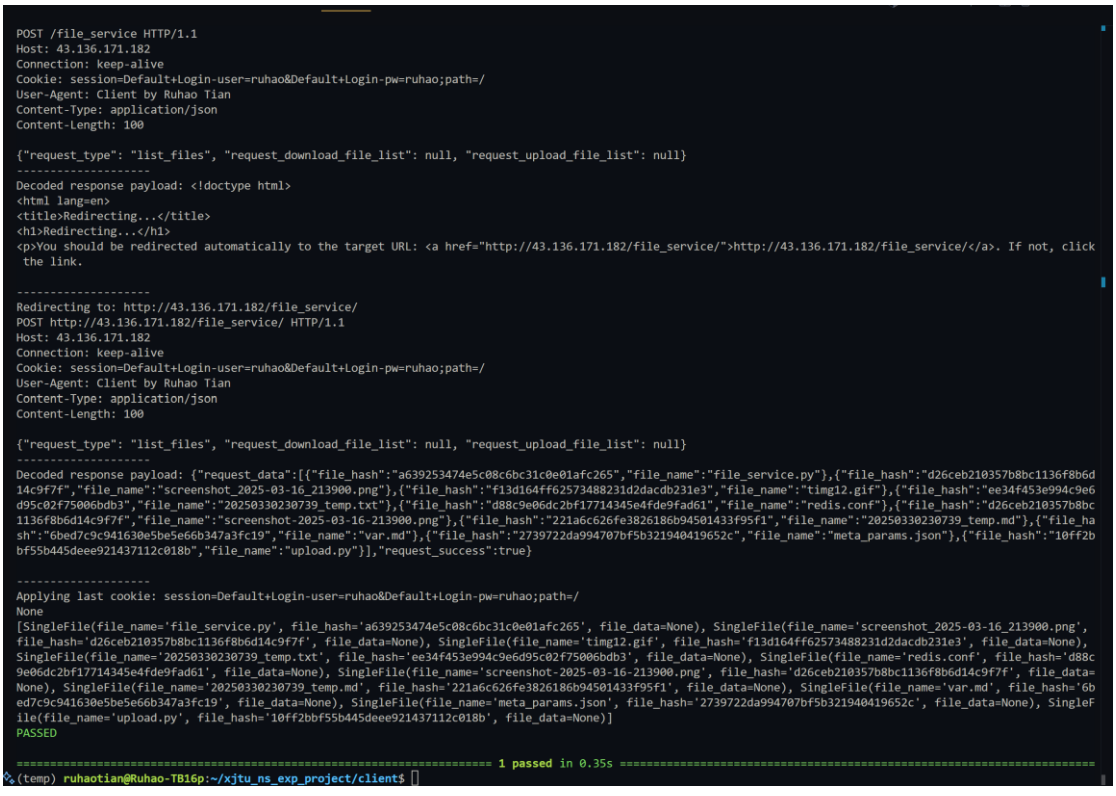
5.3.1 测试获取文件列表

先使用认证服务登陆，然后从响应状态获取 Cookie，在不进行再次登陆的情况下获取文件列表——间接测试了 Session 机制。

```
def test_fetch_file_list():
    credential = load_credentials(
        load_test_data("./testing/test_file_service/test_file_service_cases.yaml")
    )
    session = perform_login_and_get_session(credential)
```

```
    print(session)
    assert session is not None
    # for debugging
    # session.session_token = None
    http_client = HttpClientSocket()
    file_service = FileService(http_client, LocalFileBackend())
    setting = Setting()
    setting.http_request_template = DEFAULT_HTTP_REQUEST_TEMPLATE
    request_interface = FetchServerFileInterface(
        current_session=session,
        setting=setting,  # Assuming default settings are used
    )
    response = file_service.fetch_server_file_list(request_interface)
    print(response.error_message)
    print(response.file_list)
    assert isinstance(response, ServerFileList)
    assert response.valid_list == True
```

可见成功获取文件列表，并且如果仔细检查其 JSON 嵌套格式，和前文中预定义的 API 数据类结构是一一对应的：



5.3.2 测试上传/下载数据及缓存

测试逻辑基本一致：

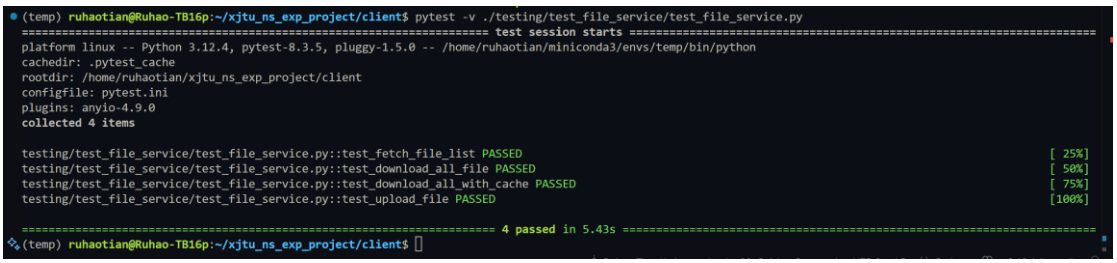
- 读取本地/服务器文件夹，清空接收方文件夹
- 上传/下载内容，再次查看接收方文件夹，确保文件到达

- 如果测试缓存，则再进行一次传输，查看这次实际传输的文件数是否为空。

具体实现请参阅源码。

如果打开输出语句，会出现大量字符编码，这是经过 Base64 编码的传输文件经过解码后的报文。

关闭输出语句，给出相关用例的测试结果：



5.4 UI 测试

UI 测试环节在现场验收中已经详细展示。

6 总结及心得体会

我们利用这次机会体验了一下迷你版的前后端全栈开发流程，并了解了事件驱动软件模型和网络服务的工作原理。尽管付出了大量时间精力，但收获颇丰。

7 附录

7.1 源码

源码可在 https://github.com/RuhaoT/xjtu_ns_exp_project.git 获得。该仓库在验收结束后即开放下载，造成不便请谅解。