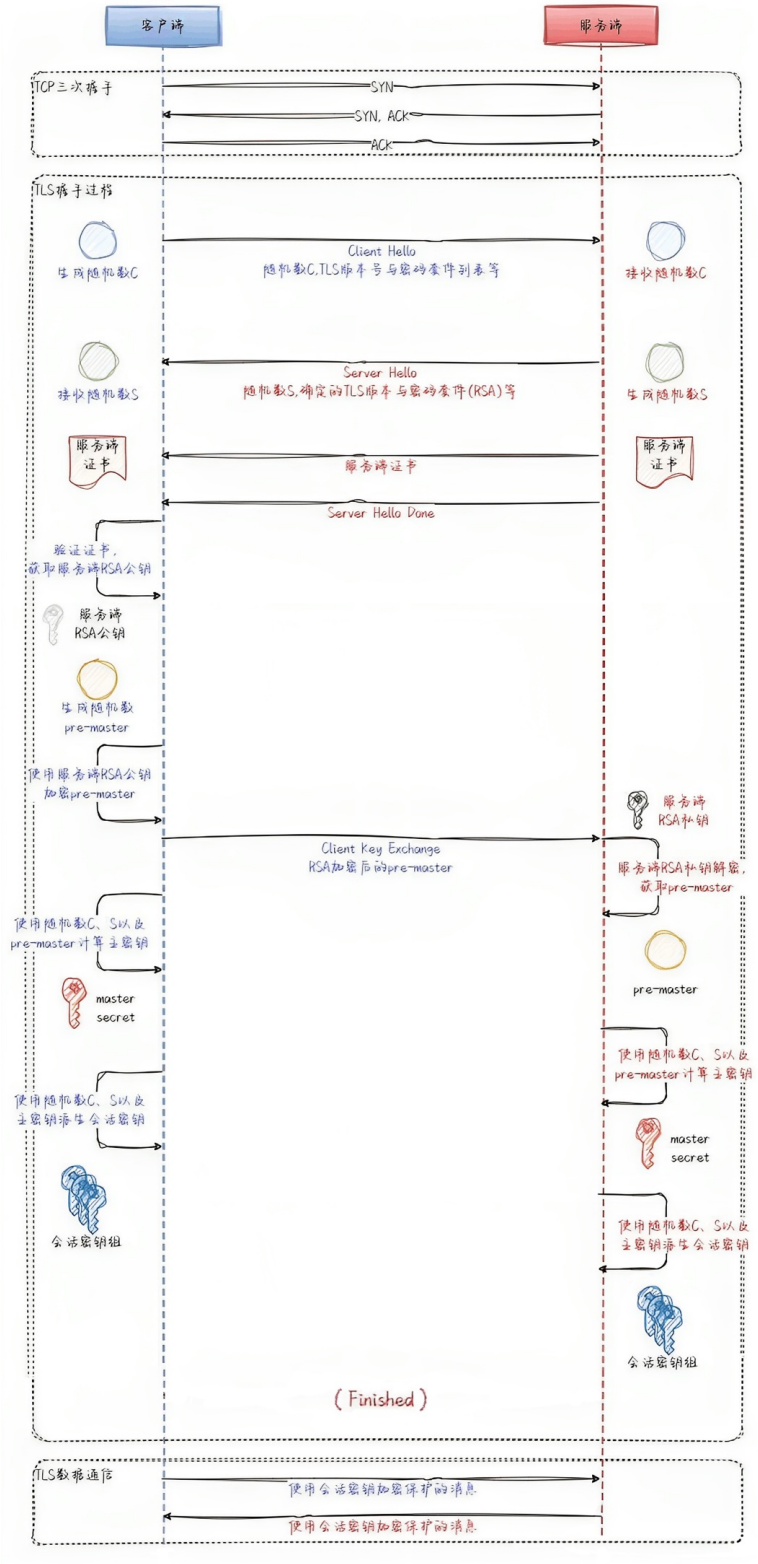
Design Document of myTLS

杨乙　21307130076　信息安全

流程概述

本项目的程序执行流程图如下：



程序架构

`server.py` 和 `client.py` 入口函数如下：

```python
# server.py

def main():
    server = Server()

    # 建立连接
    server.Server_connect()

    # TLS 握手
    server.Server_hello()
    server.Server_send_crt()
    server.Server_generate_sessionkey()

    # 加密通信
    server.Server_receive()
    server.Server_send()

    # 关闭连接
    server.Server_close()
```

```python
# client.py

def main():
    client = Client()

    # 建立连接
    client.Client_connect()

    # TLS 握手
    client.Client_hello()
    client.Client_verify_crt()
    client.Client_key_exchange()

    # 加密通信
    client.Client_send()
    client.Client_receive()

    # 关闭连接
    client.Client_close()
```

下面将对这些函数进行逐一分析。较为关键的部分会给出代码，其余代码和注释见项目文件

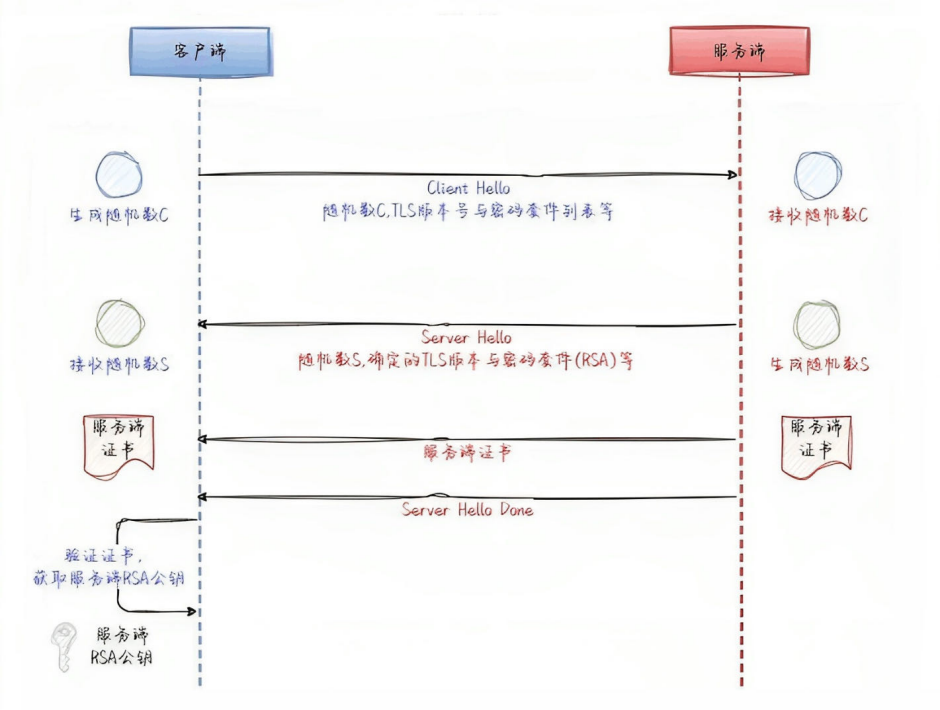
客户端和服务器连接

因为实现较为简单，这里仅给出实现原理：

1. 客户端通过 `socket()` 函数创建套接字，通过 `connect()` 函数初始化服务器连接；之后 TLS 握手信息的发送通过 `send()` 函数实现，接收通过 `recv()` 函数实现
2. 服务器通过 `socket()` 函数创建套接字，通过 `bind()` 函数绑定地址和端口，通过 `listen()` 函数监听端口，通过 `accept()` 函数被动接受连接。同样地，之后 TLS 握手信息通过 `send()` 函数发送，通过 `recv()` 函数接收。二者都使用 `close()` 函数关闭连接

初步握手

这一阶段的程序执行流程图如下：



1. 首先客户端调用 `Client_hello()` 函数，向服务器发送产生的随机数 `clientRandom`、TLS 版本号、密码套件列表等（密码套件列表在这里做了简化处理，仅提供一套密码套件 `TLS_RSA_WITH_DES_SHA256`，说明密码交换使用 RSA 算法，信息加密使用 DES 算法，MAC 生成与校验使用 SHA256 算法），并且将上述信息打印到终端

2. 服务器调用 `Server_hello()` 函数接收 `client hello` 消息，向客户端返回服务器随机数 `serverRandom`、确定的 TLS 版本和确定的密码套件，并且将上述信息打印到终端

3. 服务器调用 `Server_send_crt()` 函数，向客户端发送服务器端证书和 RSA 公钥。在实际情况中 RSA 公钥应该从证书当中提取，但如果这样需要引入 `ssl` 库，相当于用现成的库实现了 TLS。因此用 `rsa` 库生成公钥和私钥，用 Linux 的 `openssl` 指令生成证书，将二者生成字典结构，转化为字符串并发送给客户端

```python
def Server_send_crt(self):
    print('\n================ Server Send Certificate ================')
    self.serverKeys = rsa.newkeys(256)                                  # 生成公钥和私钥
    serverPubkey = self.serverKeys[0].save_pkcs1()                      # 转化为 pkcs1 格式便于传输
    serverCrt = open('server.crt').read()                               # 读取服务器证书
    Msg_server_send_crt = {'serverPubkey': serverPubkey,
                           'serverCrt': serverCrt}
    self.connectionSocket.send(str(Msg_server_send_crt).encode())   # 将字典转化为字符串传输
    # ......
```

4. 客户端调用 `Client_verify_crt()` 函数接收服务器的 `server hello` 消息和证书消息，调用 `ast.literal_eval()` 函数将这两个字符串转化回字典，并通过索引获取服务器随机数、服务器公钥、服务器证书字符串。在证书验证阶段，首先将证书字符串写入创建的 `get_server.crt` 文件，再使用 `os.system()` 运行命令 `openssl verify -CAfile ../ca/ca.crt ./get_server.crt` 来验证证书（验证通过返回 0，不通过返回 1），若不通过，调用 `Client_close()` 关闭连接，退出程序

```python
def Client_verify_crt(self):
    # ......
    print('(2) Verify certificate')
    # 将证书字符串写入创建的文件
    with open('get_server.crt', 'wb') as f:
        f.write(serverCrt.encode())
    # 运行 openssl 命令验证证书
    if os.system("openssl verify -CAfile ../ca/ca.crt ./get_server.crt") != 0:
        print('    Certificate verify fail')        # 不通过
        self.Client_close()
        exit(1)
    else:
        print('    Certificate verify pass')        # 通过
```

在这一阶段服务器的运行结果如下：

```
============== Server Hello ==============
(1) Receive client hello message from client
    clientRandom: b'\x97\xb9\xf5\xa4'
(2) Generate server random
(3) Send server hello message to client
    TLS Version: myTLS
    Cipher Suite: TLS_RSA_WITH_DES_SHA256
    Server Random: b'1\x1a\xe0?'

============== Server Send Certificate ==============
(1) Server send certificate and pubkey
    Server public key: b'-----BEGIN RSA PUBLIC KEY-----\nMCgCIQCN+4Kj5wb8qnEYBqLvNcCimkJ4YX2eeoFLyFnaS1VoMwIDAQAB\n-----END RSA PUBLIC KEY-----\n'
    Server certificate: server.crt
```

客户端的运行结果如下：

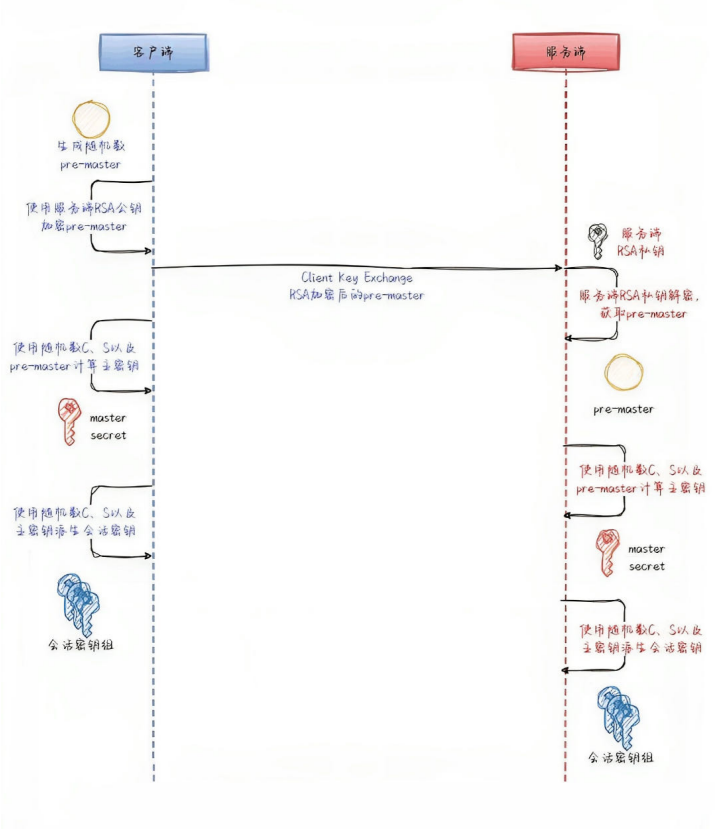
```
============== Client Hello ===============
(1) Generate client random
    clientRandom: b'\x97\xb9\xf5\xa4'
(2) Send client hello message to server
    TLS Version: myTLS
    Cipher Suite: TLS_RSA_WITH_DES_SHA256
    Client Random: b'\x97\xb9\xf5\xa4'

============== Client Verify Certificate ===============
(1) Receive server hello message from server
    serverRandom: b'1\x1a\xe0?'
(2) Receive server public key and certificate from server
    serverPubkey: b'-----BEGIN RSA PUBLIC KEY-----\nMCgCIQCN+4Kj5wb8qnEYBqLvNcCimkJ4YX2eeoFLyFnaS1VoMwIDAQAB\n-----END RSA PUBLIC KEY-----\n'
(2) Verify certificate
./get_server.crt: OK
    Certificate verify pass
```

注：本项目使用的证书由 `openssl` 生成，详见：[使用 openssl 生成证书（含openssl详解） - 南山放牧 - 博客园 (cnblogs.com)]

产生会话密钥

这一阶段的程序执行流程图如下：



1. 客户端验证服务器证书、取得服务器公钥以后，调用 `Client_key_exchange()` 函数生成 48 位预主密钥，使用服务器的 RSA 公钥加密后发送给服务器

2. 接下来客户端和服务器分别调用 `Client_generate_sessionkey()` 函数和 `Server_generate_sessionkey()` 函数来生成会话密钥。服务器端要接收客户端的加密信息，通过 RSA 私钥解密得到预主密钥。这两个函数生成会话密钥部分的实现基本相同，因此只对 `Client_generate_sessionkey()` 进行分析。首先把 `clientRandom` 和 `serverRandom` 相加作为密钥，对 `premasterSecret` 进行 MD5 哈希运算生成主密钥。接下来采用 `HKDF` 算法，通过 `hkdf_extract()` 函数将 `masterSecret` 加盐伪随机化得到 `PRK`，再使用 `hkdf_expand()` 函数将 `PRK` 扩展到 8 字节

```python
def Client_generate_sessionkey(self, premasterSecret):
    # 把 clientRandom 和 serverRandom 相加作为密钥，对预主密钥进行 MD5 哈希运算生成主密钥
    masterSecret = hmac.new(self.clientRandom + self.serverRandom,
                            premasterSecret, 'MD5').digest()
    salt = 'yangyi'.encode()
    PRK = hkdf.hkdf_extract(salt, masterSecret)     # 加盐伪随机化
    sessionKey = hkdf.hkdf_expand(PRK, b'', 8)      # 扩展到 8 字节
    return sessionKey
```

在这一阶段服务器的运行结果如下：

```
================ Server Generate Session Key ================
(1) Receive encrypted premaster secret from client
(2) Decrypt premaster secret with server private key
    premasterSecret: b'\x00\x00\xcc\x18\x14\xa1-\x0e'
(3) Generate master secret b"\xd57z`aaY\x0cpAC\xc7'\xb3/\xaf"
    masterSecret: b"\xd57z`aaY\x0cpAC\xc7'\xb3/\xaf"
(4) Generate session key
    sessionKey: b'.\xac\xf6i\x07Hl+'
```

客户端的运行结果如下：

```
================ Client Key Exchange ================
(1) Generate premaster secret
    premasterSecret: b'\x00\x00\xcc\x18\x14\xa1-\x0e'
(2) Encrypt premaster secret with server public key
(3) Send encrypted premaster secret to server

================ Client Generate Session Key ================
(1) Generate master secret
    masterSecret: b"\xd57z`aaY\x0cpAC\xc7'\xb3/\xaf"
(2) Generate session key
    sessionKey: b'.\xac\xf6i\x07Hl+'
```

加密和解密、MAC 及其验证

服务器和客户端分别调用发送函数、接收函数来实现双向传输，在发送前要对信息加密，接收后要对信息解密。因为服务器和客户端对这些函数的实现基本相同，因此仅分析客户端向服务器的发送过程：

1. 客户端在调用 `Server_send()` 发送消息前，要调用 `Encrypt()` 函数对信息进行加密，以 8 字节会话密钥作为加密密钥和填充字符，加密参数选择 `pyDes.ECB`，填充模式选择 `pyDes.PAD_PKCS5`，初始化一个 des 对象并进行加密；对加密后的密文进行 SHA256 加密得到 MAC；将密文和 MAC 拼接后再次进行 DES 加密得到最终的加密消息，最后返回加密消息和 MAC

```python
def Encrypt(self, text):
    key = self.sessionKey
    # 初始化一个 des 对象并进行加密
    des = pyDes.des(key, pyDes.ECB, key, padmode=pyDes.PAD_PKCS5)
    ciphertext = des.encrypt(text.encode())
    # 对加密后的密文进行 SHA256 加密得到 MAC
    MAC = hashlib.sha256(ciphertext).digest()
    # 将密文和 MAC 拼接后再次进行 DES 加密得到最终的加密消息
    ciphertext += MAC
    ciphermsg = des.encrypt(ciphertext)
    return ciphermsg, MAC
```

2. 服务器调用 `Server_receive()` 接收消息后，要先调用 `Decrypt()` 进行解密，获取 MAC。首先传递和客户端相同的参数初始化一个 des 对象，使用这个对象进行解密。截取后 256 位得到 MAC，其余位数为 `ciphertext`，再对 `ciphertext` 解密得到原来的消息

```
def Decrypt(self, ciphermsg):
    key = self.sessionKey
    # 传递和客户端相同的参数初始化一个 des 对象
    des = pyDes.des(key, pyDes.ECB, key, padmode=pyDes.PAD_PKCS5)
    plainmsg = des.decrypt(ciphermsg)
    ciphertext = plainmsg[0:-32]                # 截取后 256 位得到 MAC
    MAC = plainmsg[-32:len(plainmsg)]           # 其余位数为 ciphertext
    text = des.decrypt(ciphertext).decode()     # 再对 ciphertext 解密得到原来的消息
    return ciphertext, text, MAC
```

接下来服务器对 MAC 进行验证。将提取出来的 `ciphertext` 进行 SHA256 加密得到 `verifyMAC`，验证它和 MAC 是否相同即可

```
def Server_receive(self):
    # ......
    # 进行 SHA256 加密得到 verifyMAC，验证它和 MAC 是否相同
    verifyMAC = hashlib.sha256(ciphertext).digest()
    if verifyMAC != MAC:
        print('    Verification fail\nServer closing...')
        self.Server_close()
    else:
        print('    Verification pass')
        print('    text:', text)
```

在这一阶段服务器的运行结果如下：

```
=============== Server Receive ===============
(1) Server receive text from client
    MAC: b'\xf9\xf6,\\\x0b_\xec<\x05\xf4\xcb~\x9a\x08\xfb@\xbf\xc8\x9e-\xe6%A?\x851\xf1\xbeS\xe0\xc4q'
(2) Server verify MAC
    verifyMAC: b'\xf9\xf6,\\\x0b_\xec<\x05\xf4\xcb~\x9a\x08\xfb@\xbf\xc8\x9e-\xe6%A?\x851\xf1\xbeS\xe0\xc4q'
    Verification pass
    text: The long snake passes under a large elephant

=============== Server Send ===============
(1) Server send cipher message to client
    text: The quick brown fox jumps over a lazy dog
    MAC: b'\x84\xd5\xe9=\x08\x95i\x88z\xd1f*%9\x8c\xd5\xd6\x91\x05\xb6~\x8ek\x95\x15\xc0\x96\xf7\xaf\xd1\xd8\r'
```

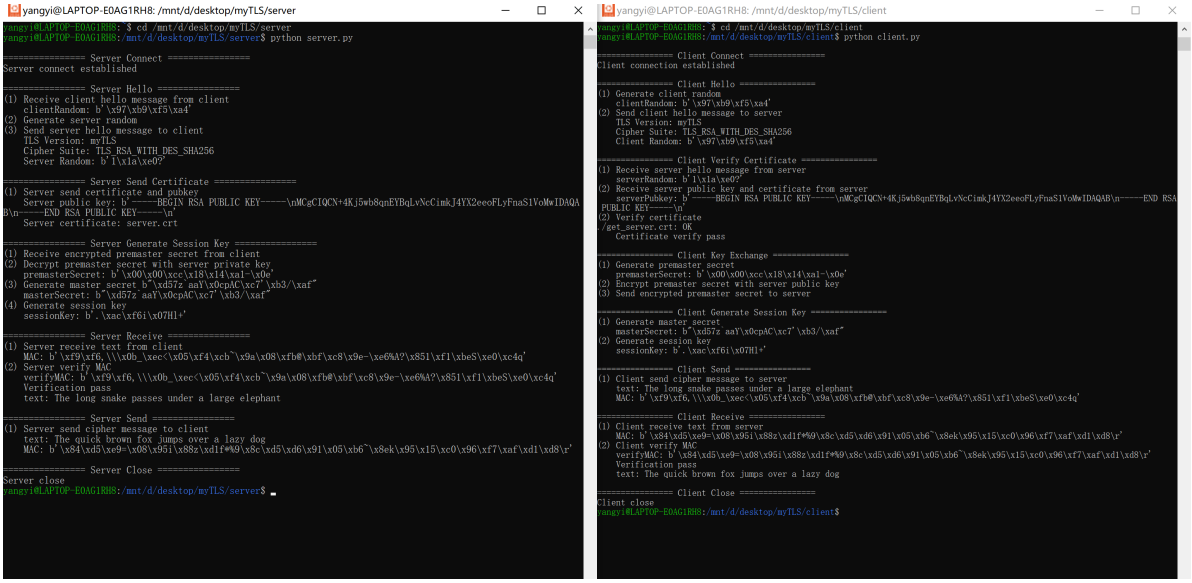
客户端的运行结果如下：

```
=============== Client Send ===============
(1) Client send cipher message to server
    text: The long snake passes under a large elephant
    MAC: b'\xf9\xf6,\\\x0b_\xec<\x05\xf4\xcb~\x9a\x08\xfb@\xbf\xc8\x9e-\xe6%A?\x851\xf1\xbeS\xe0\xc4q'

=============== Client Receive ===============
(1) Client receive text from server
    MAC: b'\x84\xd5\xe9=\x08\x95i\x88z\xd1f*%9\x8c\xd5\xd6\x91\x05\xb6~\x8ek\x95\x15\xc0\x96\xf7\xaf\xd1\xd8\r'
(2) Client verify MAC
    verifyMAC: b'\x84\xd5\xe9=\x08\x95i\x88z\xd1f*%9\x8c\xd5\xd6\x91\x05\xb6~\x8ek\x95\x15\xc0\x96\xf7\xaf\xd1\xd8\r'
    Verification pass
    text: The quick brown fox jumps over a lazy dog
```

运行结果

全部运行结果如下：



Heartbleed 漏洞

Heartbleed 是出现在加密程序库 OpenSSL 的安全漏洞。服务器或客户端如果使用的是存在缺陷的 OpenSSL 实例，则可能因此而受到攻击。漏洞产生的原因是在实现 TLS 的 Heartbeat 扩展时输入缺少边界检查