بسمه تعالی

شبکه‌های کامپیوتری

پروژه پایانی

استاد جعفری

مهتا فطرت - فاطمه السادات موسوی

بهمن ماه 1401

شرح پروژه

در این پروژه هدف پیاده سازی یک Proxy شامل یک Client و Server است که میان تعدادی Client app و Server app قرار می‌گیرند. وظیفه‌ی سمت Client این Proxy این است که بر روی یک پورت لوکال UDP از طریق سوکت گوش دهد و بسته‌هایی را که به آن ارسال می‌شوند، دریافت کرده و Header بسته‌های دریافتی را تغییر دهد؛ مثلاً به آن آدرس IP و پورت Client app و Server app را بیفزاید. سپس بسته‌ی ساخته شده را از طریق یک پورت لوکال سوکت TCP به سمت Proxy Server ارسال کند.

سمت Server، بر روی یک پورت TCP در حال گوش دادن است تا بسته‌های دریافتی از سمت Client را دریافت کرده، تغییرات ایجاد شده در Header بسته را حذف نموده و سپس بسته را از طریق یک اتصال UDP برای Server app مربوطه بفرستد.

همچنین سمت Server روی یک پورت لوکال UDP گوش می‌دهد تا در صورت دریافت پاسخ مربوط به درخواست یک Client app از سوی Server app، پس از ایجاد تغییرات مورد نیاز بر روی Header آن، بسته را از طریق اتصال TCP به سمت کلاینت Proxy برساند تا نهایتاً سمت Client با حذف این Header پاسخ Server app را استخراج کرده و به Application مربوطه در سمت Client برساند.

در این پروژه با هدف ایجاد ارتباط امن میان سمت Client و سمت Proxy Server ارتباط بین این دو به صورت TLS پیاده سازی شده است.

همچنین در این پروژه لازم است در Client و Server ارسال و دریافت به صورت موازی بتوانند کار کنند. بنابراین جهت ایجاد قابلیت خواندن و نوشتن به صورت همزمان و نیز Handle کردن درخواست‌های مربوط به چندین Client app به صورت همزمان از Multithreading و Multiprocessing در پیاده سازی استفاده شده است.

پیاده‌سازی

ساختار

ماژول xclient.py، آدرس تمام Client app ها و Server app های متناظرشان را در ابتدا به صورت آرگومان ورودی دریافت می‌کند. به ازای هر جفت Client و Server، یک socket از نوع UDP و دو socket از نوع TCP ایجاد می‌شود. سوکت UDP پیام‌های Client app را دریافت می‌کند و socket های TCP با اتصال به XServer، مبادله‌ی پیام‌های Client app و Server app را به عهده دارند. به این ترتیب که یک TCP Connection، برای ارسال پیام‌های Client app به Server app و دیگری برای ارسال پاسخ‌های Server app به Client app مورد استفاده قرار می‌گیرد. لازم به ذکر است که نوع هر اتصال TCP میان XClient و XServer، به صورت قراردادی، در اولین پیام برای XServer ارسال می‌شود.

در ادامه، به ازای هر کدام از این جفت‌های Client و Server، یک پردازه‌ی جدید ایجاد می‌شود که مبادله‌ی پیام‌های بین آن دو را به عهده خواهد داشت. در هر پردازه، دو ریسه ایجاد می‌شود. یک ریسه در یک حلقه‌ی بی‌نهایت، روی پورت UDP گوش می‌کند و پیام‌های Client app را دریافت

کرده و روی یک اتصال TCP برای XServer ارسال می‌کند و ریسه‌ی دیگر، روی یک حلقه‌ی بی‌نهایت روی اتصال TCP دیگری گوش می‌کند و پیام‌های Server app را از طریق پورت UDP به Client app ارسال می‌کند.

پیام‌های مبادله شده میان Xclient و XServer، فرمت مشخصی دارند. این پیام‌ها در بخش Header، آدرس IP و port هر دوی Client app و Server app را شامل می‌شوند. این Header به منظور تشخیص هویت اتصال‌های TCP در سمت XServer مورد استفاده قرار می‌گیرد. بخش Payload پیام‌ها نیز عیناً شامل پیام‌های Client app و Server app می‌باشد. اولین کاراکتر line-feed به عنوان delimiter بخش Header و Payload قرارداد شده‌است.

در سمت دیگر، ماژول xserver.py، در یک حلقه‌ی بی‌نهایت، منتظر دریافت اتصال‌های TCP می‌شود. هرگاه هر دو اتصال فرستنده و گیرنده‌ی یک زوج Client app و Server app دریافت شوند، یک پردازه‌ی مجزا برای مبادله‌ی پیام‌های بین آن دو و یک سوکت UDP برای دریافت پاسخ‌های Server app در XServer ایجاد می‌شود. در هر پردازه، دو ریسه‌ی اجرایی وجود دارند. یک ریسه در یک حلقه‌ی بی‌نهایت روی پورت UDP گوش می‌کند و پیام‌های Server app را روی اتصال TCP فرستنده، برای XClient ارسال می‌کند. ریسه‌ی دیگر نیز در یک حلقه‌ی بی‌نهایت، روی اتصال TCP گیرنده گوش می‌کند و پیام‌های دریافتی از سمت XClient را روی پورت UDP برای Server app ارسال می‌کند.

توجه داریم که به کمک پردازه‌ها و ریسه‌های اختصاصی‌ای که برای هر زوج Client app و Server app و برای هر کدام از عملیات‌های ارسال به سرور و دریافت از سرور ایجاد شده‌اند، این

عملیات‌ها به صورت کاملاً موازی قابل انجام هستند. همچنین لازم به ذکر است که این توازی، در ذات مسئله مورد نیاز است. چرا که Server app ها لزوماً به ازای هر پیام Client app، یک پاسخ ارسال نمی‌کند. بنابراین نمی‌توان پاسخ‌های Server را به صورت سریال با درخواست‌های Client دریافت کرد.

در ادامه، به کاربرد ماژول‌های client.py و server.py پرداخته می‌شود. این ماژول‌های کمکی، به منظور تست صحت عملکرد Proxy مورد استفاده قرار می‌گیرند و عملکرد Client app و Server app را شبیه‌سازی می‌کنند. ماژول Client در ابتدا یک پورت UDP آزاد را انتخاب کرده و آن را چاپ می‌کند. سپس با دریافت ورودی "start" از stdin، سه پیام نمونه برای XClient ارسال می‌کند. سپس روی پورت UDP گوش می‌کند تا پاسخ‌های Server app را از طریق XClient دریافت و در stdout چاپ کند.

ماژول Server نیز به طور مشابه، یک پورت UDP آزاد را انتخاب کرده و روی آن گوش می‌کند تا پیام‌های ارسالی Client را از طریق XServer دریافت کند. سپس این server app، یک عملیات ساده روی پیام دریافتی انجام می‌دهد (Uppercase کردن تمام letter ها) و پاسخ خود را روی پورت UDP برای XServer ارسال می‌کند تا برای Client app فرستاده شوند.

در انتهای این بخش، ماژول utils معرفی می‌شود. این ماژول شامل یک‌سری تابع است که مربوط به پروتکل‌های مورد استفاده توسط Proxy می‌شوند و مشترکا توسط XServer و XClient مورد استفاده قرار می‌گیرند. از جمله‌ی این عملکردها، تجزیه‌ی پیام‌های مبادله‌شده بین

XCilent و XServer به بخش‌های Header و Payload، تجزیه‌ی Header به فیلدهای آن و ارسال و دریافت پیام برروی اتصال‌های TCP را می‌توان نام برد.

ساخت certificate سرور در ارتباط tls

به منظور ساخت certificate ارتباط tls که به صورت آرگومان ورودی تابع wrap_socket در سمت xserver نیاز است، از دستور openssl استفاده شده‌است. در ابتدا با دستور

```
openssl genrsa -out private.pem 2048
```

یک کلید خصوصی به نام private در فرمت PEM ساخته می‌شود. سپس با کمک دستور

```
openssl req -new -x509 -key private.pem -out cacert.pem -days 1095
```

certificate با نام cacert و در فرمت PEM ساخته می‌شود. حال آدرس این certificate به عنوان آرگومان cert در تابع wrap_socket سمت xserver استفاده می‌شود.

جزئیات

در این بخش جزئیات پیاده‌سازی شامل کلاس‌ها و توابع ذکر می‌شوند:

- ❖ **ماژول utils.py**

این ماژول شامل توابع کمکی برای پیاده‌سازی سایر بخش‌هاست:

1. Parse_message

این تابع، یک پیام را به عنوان ورودی می‌گیرد و آن را به دو بخش Header و Payload می‌شکند و این دو بخش را به عنوان خروجی برمی‌گرداند.

```
# take a message and split it into header and payload
def parse_message(message):
    header = message.split('\n')[0]
    payload = ''.join(message.split('\n')[1:])
    return header, payload
```

2. Parse_header

این تابع یک Header را به عنوان ورودی می‌گیرد و مقادیر IP و پورت مبدا، و IP و پورت مقصد را به ترتیب به عنوان خروجی برمی‌گرداند.

```
# take a header and return the IP and port values in this format:
# sender-ip, sender-port, receiver_ip, receiver_port
def parse_header(header):
    header_args = header.split(':')
    return header_args[0], int(header_args[1]), header_args[2], int(header_args[3])
```

3. Add_header

این تابع یک Payload و دو آدرس شامل آدرس‌های کلاینت و سرور را به عنوان ورودی دریافت می‌کند و Header ای شامل آدرس IP و پورت کلاینت و سرور می‌سازد. سپس این Header را به بخش Payload افزوده و پیامی می‌سازد و این پیام را به عنوان خروجی برمی‌گرداند.

```
# make a header including the client and server addresses
# then make a packet containing the made header and payload
def add_header(payload, client_app_addr, server_app_addr):
    header = "{}:{}:{}:{}\n".format(*client_app_addr, *server_app_addr)
    return header + payload
```

4. Acked_send

این تابع یک پیام، سوکت و سایز بافر را به عنوان ورودی دریافت کرده و پیام را از طریق سوکت داده شده ارسال می‌کند. سپس یک پیام با حداکثر طول سایز بافر را به عنوان acknowledgement دریافت می‌کند.

```
# take a message and send it through a given socket. then receive an acknowledgement up to buffer size
def acked_send(message, socket, buff_size):
    print(f"Sent tcp message '{message}'.")
    socket.send(message.encode())
    socket.recv(buff_size)
```

5. Acked_recv

این تابع یک سوکت و سایز بافر را به عنوان ورودی می‌گیرد و پیامی را به اندازه‌ی حداکثر سایز بافر از سوکت دریافت کرده و یک Acknowledgement از طریق سوکت می‌فرستد. در نهایت پیام دریافت شده را به عنوان خروجی برمی‌گرداند.

```
# take a socket and receive messages from it up to buffer size. them send an acknowledgement
def acked_recv(socket, buff_size):
    message = socket.recv(buff_size).decode()
    print(f"Received tcp message '{message}'.")
    socket.send("ACK".encode())
    return message
```

❖ ماژول client.py

در این ماژول Client app پیاده‌سازی می‌شود که بعداً برای تست Proxy مورد استفاده قرار می‌گیرد.

کلاسClient: این کلاس شامل توابعی است که در ادامه ذکر می‌شوند:

1. Init

در این تابع مقداردهی اولیه انجام می‌شود.

یک UDP socket برای ارتباط بین Client و XClient و آدرس XClient در نظر گرفته می‌شود.

همچنین تعدادی پیام نمونه برای ارسال به Server app ساخته می‌شود.

```
def __init__(self):
    # consider a udp socket for the transmissions between client app and x-client
    self.udp_socket = None
    # the IP address and port of x-client
    self.xclient_addr = None
    # initialize some sample messages to send to server app
    self.messages = ["Sample Message 1.", "Sample Message 2.", "Sample Message 3."]
```

2. Bind_udp

این تابع یک سوکت UDP برای ارتباط بین Client app و XClient ساخته و آن را به یک پورت آزاد bind می‌کند. در نهایت پورت تخصیص داده شده به سوکت را چاپ می‌کند.

```python
def bind_udp(self):
    # create the udp socket for the transmissions between client app and x-client
    self.udp_socket = socket.socket(
        family=socket.AF_INET, type=socket.SOCK_DGRAM)
    # bind the udp socket to a free port
    self.udp_socket.bind((IP, 57078))
    # get the port assigned to the udp socket and print it
    udp_socket_name = self.udp_socket.getsockname()[1]
    print(f"Listening UDP on {IP}:{udp_socket_name}")
```

3. Get_xclient_init_message

این تابع اولین پیام ارسال شده از سمت XClient از طریق سوکت UDP را که حاوی آدرس XClient است، دریافت کرده و آدرس XClient را در کلاس مقداردهی می‌کند.

```python
def __init__(self):
    # consider a udp socket for the transmissions between client app and x-client
    self.udp_socket = None
    # the IP address and port of x-client
    self.xclient_addr = None
    # initialize some sample messages to send to server app
    self.messages = ["Sample Message 1.", "Sample Message 2.", "Sample Message 3."]
```

4. Send_sample_packets

این تابع پیام‌های نمونه را از سمت Client از طریق سوکت UDP به XClient ارسال می‌کند.

```python
def send_sample_packets(self):
    for message in self.messages:
        # endode and send the sample messages to x-client through udp socket
        self.udp_socket.sendto(message.encode(), self.xclient_addr)
        print(f"Sent message '{message}'.")
```

5. Receive_responses

این تابع پاسخ‌های دریافت‌شده از سمت XClient از طریق سوکت UDP را Decode کرده و چاپ می‌کند.

```
def receive_responses(self):
    for _ in self.messages:
        # receive the server responses to each of the sample messages from x-client through udp socket
        message, _ = self.udp_socket.recvfrom(BUFF_SIZE)
        # decode and print each of the received messages
        print(f"Received message '{message.decode()}'")
```

6. Start

این تابع یک سوکت UDP برای اتصال به XClient می‌سازد. سپس پیام initialization حاوی آدرس XClient UDP را دریافت می‌کند و در ادامه ارسال و دریافت بسته‌ها را با گرفتن دستور Start در ورودی، آغاز می‌کند.

```
def start(self):
    # bind a udp socket to connect to x-client
    self.bind_udp()
    # receive an initialization message from x-client
    self.get_xclient_init_message()
    # start sending and receiving the messages by typing the 'start' command
    if input() == "start":
        self.send_sample_packets()
        self.receive_responses()
```

❖ ماژول server.py

در این ماژول Server app پیاده‌سازی می‌شود که بعداً برای تست Proxy مورد استفاده قرار می‌گیرد.

کلاس Server: این کلاس شامل توابعی است که در ادامه ذکر می‌شوند:

1. Init

در این تابع مقداردهی اولیه انجام می‌شود.

یک UDP socket برای ارتباط بین Server و XServer ساخته می‌شود.

```
def __init__(self):
    # consider a udp socket for the transmissions between server app and x-server
    self.udp_socket = None
```

2. Bind_udp

این تابع یک سوکت UDP برای ارتباط بین Server app و XServer ساخته و آن را به یک پورت آزاد bind می‌کند. در نهایت پورت تخصیص داده شده به سوکت را چاپ می‌کند.

```
def bind_udp(self):
    # create the udp socket for the transmissions between server app and x-server
    self.udp_socket = socket.socket(
        family=socket.AF_INET, type=socket.SOCK_DGRAM)
    # bind the udp socket to a free port
    self.udp_socket.bind((IP, 57013))
    # get the port assigned to the udp socket and print it
    udp_socket_name = self.udp_socket.getsockname()[1]
    print(f"Listening UDP on {IP}:{udp_socket_name}")
```

3. Build_response

این تابع یک پیام را دریافت می‌کند و پاسخ به آن را می‌سازد و در خروجی برمی‌گرداند. به عنوان یک مثال ساده فرض شده که در این‌جا Server app یک رشته را دریافت کرده و Uppercase آن را برمی‌گراند.

```
# create a sample method that builds the server responses
@staticmethod
def build_response(message):
    # this server app is supposed to convert the received messages to uppercase
    # and return it to the client app
    return message.upper()
```

4. Handle_requests

این تابع در یک حلقه‌ی بی‌نهایت پیام‌ها را از XServer از طریق سوکت UDP دریافت می‌کند و آن‌ها را چاپ می‌کند. سپس پاسخ مربوط به هریک از درخواست‌ها را ساخته و پاسخ را از طریق سوکت UDP به XServer ارسال می‌کند. در انتها پاسخ‌های ساخته شده را چاپ می‌کند.

```
def handle_requests(self):
    while True:
        # receive messages from x-server up tp buffer size
        message, addr = self.udp_socket.recvfrom(BUFF_SIZE)
        # print the received message
        print(f"Received message '{message}'.")
        # build the response for each received message
        response = self.build_response(message.decode())
        # send the built response to x-server through the udp socket
        self.udp_socket.sendto(response.encode(), addr)
        # print the sent response
        print(f"Sent message '{response}'.")
```

5. Start

این تابع یک سوکت UDP برای ارسال و دریافت بسته‌ها ساخته و در ادامه با صدا

زدن تابع Handle_requests درخواست‌های ارسال شده را Handle می‌کند.

```
# start the server app job by creating a udp connection and handle the q through it
def start(self):
    self.bind_udp()
    self.handle_requests()
```

❖ ماژول xclient.py

این ماژول شامل دو کلاس ClientHandler و XClient است.

کلاس XClient: این کلاس شامل توابع زیر است:

1. Init

در این تابع آرگومان‌های ورودی با عنوان args تعریف می‌شوند.

```
def __init__(self):
    self.args = None
```

2. Parse_input_argument

این تابع آرگومان‌های مورد نیاز شامل IP و پورت Client app و Server app را از ترمینال

دریافت کرده و متغیر args را مقداردهی می‌کند.

```python
# take and parse arguments as input
def parse_input_argument(self):
    parser = argparse.ArgumentParser(
        description='This is the x-client program that creates a tunnel to the server over TCP connection.'
    )

    parser.add_argument(
        '-ut',
        '--udp-tunnel',
        action='append',
        required=True,
        help="Specify client and server app addresses. The format is 'client ip:client port:server ip:server port'."
    )

    self.args = parser.parse_args()
```

3. Start_processes

این تابع به ازای هر Client app مقادیر IP و پورت Client app و Server app را مقداردهی می‌کند. سپس با داشتن این اطلاعات یک Process به ازای هر یک از Client app ها می‌سازد که آن Client app را Handle می‌کند.

```python
# for each client app take the ip and port of client and server apps
def start_processes(self):
    for tun_addr in self.args.udp_tunnel:
        client_app_ip = tun_addr.split(':')[0]
        client_app_port = int(tun_addr.split(':')[1])
        server_app_ip = tun_addr.split(':')[2]
        server_app_port = int(tun_addr.split(':')[3])

        # for each client create a client handler process to handle it
        ClientHandler(client_app_ip, client_app_port,
                      server_app_ip, server_app_port).start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Closing the connections...")
```

4. Start

این تابع ابتدا آرگومان‌های ورودی را Parse می‌کند. سپس به ازای هر Client یک Process جدید ایجاد و اجرا می‌کند تا آن Client را Handle کند.

```
# start by parsing the input arguments and stating  processes for each client app
def start(self):
    self.parse_input_argument()
    self.start_processes()
```

کلاس ClientHandler: این کلاس شامل توابع زیر است:

1. Init

این تابع IPها و پورت‌های Client app و Server app را مقداردهی اولیه می‌کند. همچنین یک سوکت UDP برای ارتباط بین XClient و هر Client app در نظر می‌گیرد. در نهایت دو سوکت TCP یکی برای ارسال داده به XServer و دیگری برای دریافت داده به XServer در نظر می‌گیرد.

```
def __init__(self, client_app_ip, client_app_port, server_app_ip, server_app_port):
    super(ClientHandler, self).__init__()
    # initialize the client and server app ips and ports
    self.client_app_addr = client_app_ip, client_app_port
    self.server_app_addr = server_app_ip, server_app_port

    # consider a udp socket for the transmissions between each client app and x-client
    self.listening_udp_socket = None
    # consider sending and receiving tcp sockets for each client app
    self.sending_tcp_socket = None
    self.receiving_tcp_socket = None
```

2. Handle_udp_conn_recv

این تابع در یک حلقه‌ی بی‌نهایت پیام‌هایی که از طریق سوکت UDP از طرف Client app می‌آید را با حداکثر طول سایز بافر در نظر گرفته شده، دریافت می‌کند و پیام‌های دریافت شده را چاپ می‌کند. سپس با فراخواندن تابع Add_header به این پیام Headerای شامل آدرس Client app و Server app می‌افزاید و با استفاده از تابع Ack_send آن را از طریق سوکت TCP به XServer ارسال کرده و یک پیام Acknowledgement دریافت می‌کند.

```
# handle udp requests the client app
def handle_udp_conn_recv(self):
    while True:
        # receive message from the client app through the udp socket up to buffer size
        payload, _ = self.listening_udp_socket.recvfrom(ClientHandler.UDP_BUFF_SIZE)
        # print the received message
        print(f"Received udp message '{payload}'.")
        # create a message by adding header containing client and server apps' addresses
        # to the client app message
        message = add_header(payload.decode(), self.client_app_addr, self.server_app_addr)
        # send the built message through sending tcp connection and receive an
        # acknowledgement from it
        acked_send(message, self.sending_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
```

3. Handle_tcp_conn_recv

این تابع در یک حلقه‌ی بی‌نهایت روی سوکت TCP گوش می‌کند تا پیام‌های ارسال شده از سمت XServer را دریافت کند. پس از دریافت پیام بخش Payload آن را در نظر گرفته و Header اضافه شده در XServer را حذف می‌کند. در ادامه این پیام را از طریق سوکت UDP به Client app مربوطه ارسال کرده و این پیام‌های ارسال شده را چاپ نیز می‌کند.

```
# handle tcp messages from x-server
def handle_tcp_conn_recv(self):
    while True:
        # receive messages from the x-server through the tcp socket
        message = acked_recv(self.receiving_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
        # take the payload of the message and remove the added header
        _, payload = parse_message(message)
        # send the payload to the proper client app through udp socket
        self.listening_udp_socket.sendto(payload.encode(), self.client_app_addr)
        # print the sent message
        print(f"Sent udp message '{payload}'.")
```

4. Create_udp_connection

این تابع یک اتصال UDP برای Handle کردن ارتباط بین XClient و Client app می‌سازد و آن را به یک پورت آزاد با آدرس XClient IP bind می‌کند. سپس پورت در نظر گرفته شده برای سوکت UDP را گرفته و آن را از طریق سوکت UDP به Client app فرستاده و اعلام می‌نماید.

```
# create a udp connection to handle the transmissions between x-client and client app
def create_udp_connection(self):
    udp_socket_name = None
    try:
        # create a udp socket and bind it to x-client ip and a free port
        self.listening_udp_socket = socket.socket(
            family=socket.AF_INET, type=socket.SOCK_DGRAM)
        self.listening_udp_socket.bind((IP, 0))
        # take the name of the created udp socket
        udp_socket_name = self.listening_udp_socket.getsockname()[1]
        self.listening_udp_socket.sendto(str(udp_socket_name).encode(), self.client_app_addr)
    except socket.error as e:
        # handle socket exceptions
        print("(Error) Error opening the UDP socket: {}".format(e))
        print("(Error) Cannot open the UDP socket {}:{} or bind to it".format(
            IP, udp_socket_name))
    else:
        print("Bind to the UDP socket {}:{}".format(IP, udp_socket_name))
```

5. Create_sending_tcp_connection

این تابع به ازای هر Client app یک اتصال TCP برای ارسال اطلاعات از XClient به XServer ساخته و آن را با wrap tls می‌کند. در ابتدا پیامی حاوی Payload Sending و Header ساخته شده شامل آدرس IP و پورت Client app و Server app را ساخته و به XServer ارسال می‌کند که در واقع نوع اتصال را مشخص می‌کند.

```
# create tcp connection for each client app to send messages to x-server
def create_sending_tcp_connection(self):
    try:
        # make a tcp socket
        self.sending_tcp_socket = socket.socket()
        self.sending_tcp_socket = ssl.wrap_socket(self.sending_tcp_socket)  # ssl_version = PROTOCOL_TLS
        # connect the tcp socket to x-server
        self.sending_tcp_socket.connect((XSERVER_IP, XSERVER_PORT))
        # create an initialization message
        payload = "sending"
        # add a header containing client and server apps' addresses to the message
        message = add_header(payload, self.client_app_addr, self.server_app_addr)
        # send the initialization message to x-server trough the tcp sending socket
        acked_send(message, self.sending_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
    except socket.error as e:
        # handle socket exceptions
        print("(Error) Error opening the sending TCP socket: {}.".format(e))
        print("(Error) Cannot connect sending TCP socket to {}:{}.".format(
            XSERVER_IP, XSERVER_PORT))
        sys.exit(1)
    else:
        print("Connected the TCP socket to {}:{}.".format(
            XSERVER_IP, XSERVER_PORT))
```

6. Create_receiving_tcp_connection

این تابع به ازای هر Client app یک اتصال TCP برای ارسال اطلاعات از XClient به XServer ساخته و آن را با wrap tls می‌کند. در ابتدا پیامی حاوی Payload Receiving و Header ساخته شده شامل آدرس IP و پورت Client app و Server app را ساخته و به XServer ارسال می‌کند.

```
# create tcp connection for each client app to receive messages from x-server
def create_receiving_tcp_connection(self):
    try:
        # make a tcp socket
        self.receiving_tcp_socket = socket.socket()
        # connect the tcp socket to x-server
        self.receiving_tcp_socket.connect((XSERVER_IP, XSERVER_PORT))
        # create an initialization message
        payload = "receiving"
        # add a header containing client and server apps' addresses to the message
        message = add_header(payload, self.client_app_addr, self.server_app_addr)
        # send the initialization message to x-server trough the tcp sending socket
        acked_send(message, self.receiving_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
    except socket.error as e:
        # handle socket exceptions
        print("(Error) Error opening the receiving TCP socket: {}".format(e))
        print("(Error) Cannot connect receiving TCP socket to {}:{}.".format(
            XSERVER_IP, XSERVER_PORT))
    else:
        print("Connected the TCP socket to {}:{}.".format(
            XSERVER_IP, XSERVER_PORT))
```

7. Run

در این تابع به ازای هر کدام از Client app ها یک سوکت UDP و یک دو سوکت TCP یکی برای نوشتن و یکی برای خواندن ساخته می‌شود. سپس دو ترد ساخته می‌شود که در یکی از آن‌ها ارتباط با Client app از طریق سوکت Handle UDP می‌شود و در دیگری ارتباط با XServer از طریق سوکت Handle TCP می‌شود.

```
def run(self):
    # for each client app process, create a udp connection
    self.create_udp_connection()

    # for each client app, create sending and receiving tcp connections
    self.create_sending_tcp_connection()
    self.create_receiving_tcp_connection()

    # handle the udp connection from client app in a thread
    Thread(target=self.handle_udp_conn_recv).start()
    # handle the tcp connection to the x-server in another thread
    Thread(target=self.handle_tcp_conn_recv).start()
```

❖ ماژول xserver.py

این ماژول شامل دو کلاس XServer و ClientHandler است.

کلاس XServer: این کلاس شامل توابع زیر است:

1. Init

در این تابع یک سوکت TCP برای گوش دادن به پیام‌هایی که از سمت XClient می‌آیند، ساخته می‌شود. همچنین دو دیکشنری ساخته می‌شود که اتصالات Receiving و Sending از سمت XClient را نگه می‌دارند.

```
def __init__(self):
    # consider a tcp socket for listening upcoming messages from x-client
    self.tcp_socket = None
    # create dictionaries to store the x-client's sending and receiving connections
    self.tcp_sending_conns = {}
    self.tcp_receiving_conns = {}
```

2. Identify_connection

این تابع یک اتصال TCP به عنوان ورودی می‌گیرد. اولین پیام را از طریق سوکت TCP از XClient دریافت کرده و یک پیام نیز به عنوان Acknowledgement می‌فرستد. سپس با فراخواندن تابع Parse_message پیام دریافت شده را به دو بخش Header و Payload تقسیم می‌کند و آدرس را از Parse Header می‌کند و به همراه Payload به عنوان خروجی برمی‌گرداند.

```python
def identify_connection(self, conn):
    # receive the first message from x-client trough the tcp socket
    message = acked_recv(conn, XServer.BUFF_SIZE)
    # split the message into its header and payload
    header, payload = parse_message(message)
    # obtain the connection address from th header
    conn_addr = parse_header(header)
    # return the connection address and the connection type(payload might be 'sending' or 'receiving')
    return conn_addr, payload
```

3. Connection_duplex

این تابع بررسی می‌کند که آیا هر دو اتصال Sending TCP و Receiving به ازای یک Client app خاص ایجاد شده است یا نه

```python
def connection_duplex(self, conn_addr):
    return conn_addr in self.tcp_sending_conns and conn_addr in self.tcp_receiving_conns
```

4. Handle_connections

این تابع در یک حلقه‌ی بی‌نهایت از طریق سوکت TCP به پیام‌هایی که از سمت XClient می‌آید گوش می‌کند و با فراخواندن تابع identify_connection آدرس و نوع آن را مشخص می‌کند و آن را به دیکشنری مربوطه اضافه می‌کند. سپس در صورتی که اتصالات Receiving و Sending هردو برقرار شده باشند یک Process می‌سازد تا آن Client را Handle کند.

```python
def handle_connections(self):
    while True:
        # accept connections from x-client to the tcp socket
        conn, _ = self.tcp_socket.accept()
        # identify the connection address and type of it (sending or receiving)
        conn_addr, send_recv = self.identify_connection(conn)
        # add each type of connection to the proper dictionary
        # (key= connection address, value= tcp socket)
        conn_dict = self.tcp_sending_conns if send_recv == "sending" else self.tcp_receiving_conns
        conn_dict[conn_addr] = conn

        # if there are sending and receiving connections for a specific client,
        # create a client handler process to handle it
        if self.connection_duplex(conn_addr):
            ClientHandler(*conn_addr, self.tcp_sending_conns[conn_addr],
                          self.tcp_receiving_conns[conn_addr]).start()
```

5. Create_listening_tcp_socket

این تابع یک سوکت TCP می‌سازد و آن را به آدرس IP و پورت XServer bindمی‌کند. سپس روی این سوکت گوش می‌دهد. به منظور برقراری ارتباط امن، یک SSL-context ساخته می‌شود و Certificate ای که توضیح نحوه‌ی ساخت آن پیش‌تر آمده‌است برای Context، Set کرده و سوکت را با tls، wrap می‌کند.

```
# make a tcp socket and bind it to the x-server ip and port
def create_listening_tcp_socket(self):
    self.tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.tcp_socket.bind((XSERVER_IP, XSERVER_PORT))
    print(f"Listening TCP on {XSERVER_IP}:{XSERVER_PORT}")
    self.tcp_socket.listen()
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain('/home/matt/Downloads/Semester7/network/project/cacert.pem', '/home/matt/Downloads/Semester7/network/project/private.pem')

    self.tcp_socket = context.wrap_socket(self.tcp_socket, server_side=True)
```

6. Start

این تابع برنامه را با ساخت یک سوکت TCP به منظور برقراری ارتباط با XClient آغاز کرده و در ادامه با فراخواندن تابع Handle_connections اتصالات TCP و UDP را Handle می‌نماید.

```
# start by creating a tcp socket and handling the connections
def start(self):
    self.create_listening_tcp_socket()
    self.handle_connections()
```

کلاس ClientHandler: این کلاس شامل توابع زیر است:

1. Init

این تابع IPها و پورت‌های Client app و Server app را مقداردهی اولیه می‌کند. همچنین یک سوکت UDP برای ارتباط بین XServer و هر Server app در نظر می‌گیرد. همچنین دو سوکت TCP XClient را برای ارتباط با XClient در نظر گرفته و مقداردهی می‌کند.

```
def __init__(self, client_app_ip, client_app_port, server_app_ip, server_app_port, sending_tcp_socket,
             receiving_tcp_socket):
    super(ClientHandler, self).__init__()
    # initialize the client and server app ips and ports
    self.client_app_addr = client_app_ip, client_app_port
    self.server_app_addr = server_app_ip, server_app_port
    # initialize the x-client's sending and receiving sockets
    self.sending_tcp_socket = sending_tcp_socket
    self.receiving_tcp_socket = receiving_tcp_socket

    # consider a udp socket for the transmissions between server app and x-server
    self.udp_socket = None
```

2. Handle_udp_conn_recv

این تابع در یک حلقه‌ی بی‌نهایت پیام‌هایی که از طریق سوکت UDP از طرف Server app می‌آید را با حداکثر طول سایز بافر در نظر گرفته شده، دریافت می‌کند و پیام‌های دریافت شده را چاپ می‌کند. سپس با فراخواندن تابع Add_header به این پیام Headerای شامل آدرس Client app و Server app می‌افزاید و با استفاده از تابع Ack_send آن را از طریق سوکت TCP به XClient ارسال کرده و یک پیام Acknowledgement دریافت می‌کند.

```
# handle udp requests the server app
def handle_udp_conn_recv(self):
    while True:
        # receive message from the server app through the udp socket up to the buffer size
        payload, _ = self.udp_socket.recvfrom(ClientHandler.UDP_BUFF_SIZE)
        # print the received message
        print(f"Received udp message '{payload}'.")
        # create a message by adding header containing client and server apps' addresses
        # to the server app message
        message = add_header(payload.decode(), self.client_app_addr, self.server_app_addr)
        # send the built message through  x-client's tcp connection and receive an
        # acknowledgement from it
        acked_send(message, self.receiving_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
```

3. Handle_tcp_conn_recv

این تابع در یک حلقه‌ی بی‌نهایت روی سوکت TCP گوش می‌کند تا پیام‌های ارسال شده از سمت XServer را دریافت کند. پس از دریافت پیام بخش Payload آن را در نظر گرفته و Header اضافه شده در XClient را حذف می‌کند. در ادامه این پیام را از طریق سوکت UDP به Server app مربوطه ارسال کرده و این پیام‌های ارسال شده را چاپ نیز می‌کند.

```
# handle tcp requests from x-client
def handle_tcp_conn_recv(self):
    while True:
        # receive messages from the x-client through the tcp socket
        message = acked_recv(self.sending_tcp_socket, ClientHandler.TCP_BUFF_SIZE)
        # take the payload of the message and remove the added header
        _, payload = parse_message(message)
        # send the payload to the proper server app through udp socket
        self.udp_socket.sendto(payload.encode(), self.server_app_addr)
        # print the sent message
        print(f"Sent udp message '{payload}'.")
```

4. Create_udp_connection

این تابع یک اتصال UDP برای Handle کردن ارتباط بین XServer و Server app می‌سازد و آن را به یک پورت آزاد با آدرس XServer IP bind می‌کند. سپس پورت در نظر گرفته شده برای سوکت UDP را گرفته و آن را از طریق سوکت UDP به Server app فرستاده و اعلام می‌نماید.

```
# create a udp connection to handle the transmissions between x-server and server app
def create_udp_connection(self):
    udp_socket_name = None
    try:
        # create a udp socket and bind it to x-server ip and a free port
        self.udp_socket = socket.socket(
            family=socket.AF_INET, type=socket.SOCK_DGRAM)
        self.udp_socket.bind((XSERVER_IP, 0))
        # take the name of the created udp socket
        udp_socket_name = self.udp_socket.getsockname()[1]
    except socket.error as e:
        # handle socket exceptions
        print("(Error) Error opening the UDP socket: {}".format(e))
        print("(Error) Cannot open the UDP socket {}:{} or bind to it".format(
            XSERVER_IP, udp_socket_name))
    else:
        print("Bind to the UDP socket {}:{}".format(
            XSERVER_IP, udp_socket_name))
```

5. Run

در این تابع به ازای هر کدام از Client app ها یک سوکت UDP ساخته می‌شود. سپس دو ترد ساخته می‌شود که در یکی از آن‌ها ارتباط با Server app از طریق سوکت Handle UDP می‌شود و در دیگری ارتباط با XClient از طریق سوکت TCP Handle می‌شود.

```
def run(self):
    # for each client app process, create a udp connection
    self.create_udp_connection()

    # handle the udp connection from server app in a thread
    Thread(target=self.handle_udp_conn_recv).start()
    # handle the tcp connection from x-client in another thread
    Thread(target=self.handle_tcp_conn_recv).start()
```

خروجی

به منظور تست Proxy در یک سناریوی مثالی، به ترتیب زیر عمل می‌کنیم:

1. ماژول client.py را با دستور python3 client.py اجرا می‌کنیم.
   - در نتیجه‌ی این کار، client روی یک پورت UDP آزاد سوکتی می‌سازد و IP و port مربوطه را در stdout نمایش می‌دهد.
2. ماژول server.py را با دستور python3 server.py اجرا می‌کنیم.
   - در نتیجه‌ی این کار، server روی یک پورت UDP آزاد سوکتی می‌سازد و IP و port مربوطه را در stdout نمایش می‌دهد.
3. ماژول xserver.py را با دستور python3 xserver.py اجرا می‌کنیم.
   - در نتیجه‌ی این کار، xserer روی ip لوکال و پورت ۸۰۸۰ یک سوکت TCP می‌سازد و منتظر connection می‌شود.
4. ماژول xclient.py را به کمک IP:port های خروجی در stdout برنامه‌های client.py و server.py با دستور

```
python3 xclient.py -ut c_ip:c_port:s_ip:s_port
```

اجرا می‌کنیم.

   - در نتیجه‌ی این کار، xclient، اتصال‌های UDP و TCP لازم برای client و server را برقرار می‌کند و منتظر دریافت پیام از سمت آن‌ها می‌ماند.
5. در stdin برنامه‌ی client.py، واژه‌ی start را وارد می‌کنیم.

- در نتیجه‌ی این کار، client سه پیام نمونه برای xclient ارسال می‌کند. سپس با فاصله‌ی کمی، پاسخ این پیام‌ها را (که uppercase شده‌ی پیام‌های ارسالی است)، دریافت می‌کند و در خروجی نمایش می‌دهد. این مورد در تصویر زیر قابل مشاهده است.

```
matt@matt-X542URR:~/Downloads/Semester7/network/project$ python3 client.py
Listening UDP on localhost:39022
start
Sent message 'Sample Message 1.'.
Sent message 'Sample Message 2.'.
Sent message 'Sample Message 3.'.
Received message 'SAMPLE MESSAGE 1.'
Received message 'SAMPLE MESSAGE 2.'
Received message 'SAMPLE MESSAGE 3.'
```

همچنین پیام‌های دریافتی و پاسخ‌های ارسالی توسط server app در تصویر زیر آمده‌است.

```
matt@matt-X542URR:~/Downloads/Semester7/network/project$ python3 server.py
Listening UDP on localhost:39437
Received message 'b'Sample Message 1.''.
Sent message 'SAMPLE MESSAGE 1.'.
Received message 'b'Sample Message 2.''.
Sent message 'SAMPLE MESSAGE 2.'.
Received message 'b'Sample Message 3.''.
Sent message 'SAMPLE MESSAGE 3.'.
```

اما فرمت برخی پیام‌های ارسالی و دریافتی در xclient و xserver متفاوت است. چرا که این دو موجودیت، پیام‌هایی را در قالب قراردادی proxy دریافت و ارسال می‌کنند. همانطور که در تصویر زیر مشاهده می‌شود، xserver ابتدا با اتصالTCP اول، نوع آن و آدرس client app و server app را دریافت می‌کند. سپس با اتصال TCP دوم، نوع آن (یعنی receiving) و Header مشابه را دریافت کرده‌است. سپس یک port از نوع UDP برای server app ایجاد کرده‌است.

در ادامه، xserver سه پیام ارسالی client را در قالب پیام‌های Proxy (شامل Header)، دریافت کرده و پاسخ این پیام‌ها را در خارج از این قالب، از server دریافت کرده و با افزودن header به xclient ارسال کرده‌است.

```
pymatt@matt-X542URR:~/Downloads/Semester7/network/project$ python3 xserver.py
Listening TCP on localhost:8080
Received tcp message 'localhost:39022:localhost:39437
sending'.
Received tcp message 'localhost:39022:localhost:39437
receiving'.
Bind to the UDP socket localhost:34411
Received tcp message 'localhost:39022:localhost:39437
Sample Message 1.'.
Sent udp message 'Sample Message 1.'.
Received tcp message 'localhost:39022:localhost:39437
Sample Message 2.'.
Received udp message 'b'SAMPLE MESSAGE 1.''.
Sent tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 1.'.
Sent udp message 'Sample Message 2.'.
Received tcp message 'localhost:39022:localhost:39437
Sample Message 3.'.
Sent udp message 'Sample Message 3.'.
Received udp message 'b'SAMPLE MESSAGE 2.''.
Sent tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 2.'.
Received udp message 'b'SAMPLE MESSAGE 3.''.
Sent tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 3.'.
```

در نهایت خروجی سمت xclient را در تصویر زیر مشاهده می‌کنید. این موجودیت ابتدا یک سوکت UDP روی یک پورت آزاد ایجاد کرده و آن را اعلام کرده‌است. سپس دو اتصال TCP با xserver برقرار کرده‌است که برای هر کدام، نوع آن (receiving/sending) را طبق پروتکل به xserver اعلام کرده‌است. همچنین این پیام‌ها شامل Header پروکسی هستند که آدرس آی‌پی و پورت client app و server app مربوطه را در بر دارند. سپس xclient پیام‌های ارسالی client را بدون Header دریافت کرده و با افزودن Header، آن را برای xserver ارسال کرده‌است.

```
pymatt@matt-X542URR:~/Downloads/Semester7/network/project$ python3 xclient.py -ut localhost:39022:localhost:39437
Bind to the UDP socket localhost:53560
/home/matt/Downloads/Semester7/network/project/xclient.py:88: DeprecationWarning: ssl.wrap_socket() is deprecated, use SSLContex
t.wrap_socket()
  self.sending_tcp_socket = ssl.wrap_socket(self.sending_tcp_socket)  # ssl_version = PROTOCOL_TLS
Sent tcp message 'localhost:39022:localhost:39437
sending'.
Connected the TCP socket to localhost:8080.
/home/matt/Downloads/Semester7/network/project/xclient.py:112: DeprecationWarning: ssl.wrap_socket() is deprecated, use SSLConte
xt.wrap_socket()
  self.receiving_tcp_socket = ssl.wrap_socket(self.receiving_tcp_socket)
Sent tcp message 'localhost:39022:localhost:39437
receiving'.
Connected the TCP socket to localhost:8080.
Received udp message 'b'Sample Message 1.''.
Sent tcp message 'localhost:39022:localhost:39437
Sample Message 1.'.
Received udp message 'b'Sample Message 2.''.
Sent tcp message 'localhost:39022:localhost:39437
Sample Message 2.'.
Received udp message 'b'Sample Message 3.''.
Sent tcp message 'localhost:39022:localhost:39437
Sample Message 3.'.
Received tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 1.'.
Sent udp message 'SAMPLE MESSAGE 1.'.
Received tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 2.'.
Sent udp message 'SAMPLE MESSAGE 2.'.
Received tcp message 'localhost:39022:localhost:39437
SAMPLE MESSAGE 3.'.
Sent udp message 'SAMPLE MESSAGE 3.'.
```