《计算机网络第八次作业》

班级：信安 2302 班

学号：202308060227

姓名：石云博

目录

一.问题描述

Assignment 8: Fast! Fast! Fast!

1. Assuming the underlying network layer uses IP protocol, try design a transfer layer protocol that is suitable for Interstellar communications.

2.Assuming you have total control of the transfer layer, the network layer, and the link layer, design a set of protocols that is suitable for the Inter-continental optic fiber link. How could you send and receive data efficiently between Hongkong and Los Angelos?

3. Design and implement a program that could transfer large files (1GB or bigger)

between laptop computers using the campus network through campus WiFi access. Test it and analyze the outcome.

二.问题分析

首先，通过查看 nasa 网站，我们可以知道，星际通信面临极高时延（数分钟到数小时）、频繁中断和带宽受限，需要采用**Delay/Disruption-Tolerant Networking（DTN）**思想，结合“Bundle Protocol”进行消息打包、存储转发和托管转移。

港美洲际光纤链路在~13 000 km 下往返时延约 130 ms，可利用波分复用（DWDM）、前向纠错（FEC）、**大帧（Jumbo Frame）以及高效拥塞控制（如 BBR/CUBIC）等技术，并在端点部署协议加速器（PEP）**以充分利用链路容量。

校园 WiFi 大文件传输要应对丢包率高、吞吐量抖动，需要用并行/多路 TCP 或基于 UDP 的自定义可靠传输，结合分片、并发传输、重传机制和校验，并用 Python sockets + 多线程快速实现。

对于星际通信传输层协议设计：我们需要考虑到的问题是：

超高时延：火星最优位置与地球距离约 5 光分钟，最差可达 20 光分钟，RTT 可达 40 分钟以上。

频繁中断：行星自转、轨道运动、太阳遮挡等都会导致通信链路时断时续。

带宽受限：深空链路常用 X 波段/Ka 波段，带宽有限且误码率较高。

不适用 TCP/IP 端到端模型：传统 TCP 对超时丢包敏感，会陷入拥塞控制退避。

初步设想，可以实现的构建有下面几种：

首先，分层结构：

应用层：科考任务数据、高分辨率图像、遥测等。

捆绑层 (Bundle Layer)：实现“消息” (bundle) 打包、存储转发、托管转移 (custody transfer)。

其次，对于 RFC 编辑器

底层传输：可选 TCP、UDP 或物理链路专有协议。

Bundle Protocol（RFC 5050)：

打包：将若干应用层数据封装为一个 bundle，附带到期时间、优先级、路由信息。存储转发：中间节点持久化存储 bundle，直到下一个跃点可达后再转发。托管转移：节点间明确责任，成功转发后前节点可删本地副本。

拥塞与流量控制：

端到端不做窗口控制，中间节点根据存储资源和带宽作本地决策。可选“接力式拥塞”：逐跳调整发送速率，避免一次性洪泛。

差错与安全：

对 bundle 头与数据块做 CRC/数字签名。支持跨行星加密与鉴权。

对于校园 WiFi 大文件（≥1 GB）传输程序设计与测试，我们进行了如下的问题分析：

首先，对于 WiFi 特性来说，其具有丢包率高、带宽抖动、切换漫游可能短断种种问题。

第二文件体积大：一次性发送风险大，易因中断重传全部数据。

第三个问题是校园网络是多用户共用 AP，可能带宽受限。

基于以上的内容，我们进行传输协议选型

第一，基于 TCP 的分片多路并行传输

将文件拆成 N 个子文件（chunk)，每个 chunk 由单独 TCP 连接并行传输。

参考项目：prathampt/parallelFileTransfer，分片并行提高吞吐。

第二，基于 UDP 的自定义可靠传输

用固定大小（如 4 KB）chunk，通过 UDP 发送，并在应用层实现 ACK、超时重传。

参考 StackOverflow 示例：UDP 传文件，使用分界符或序号重组。

第三，混合方案

控制连接用 UDP 快速发送，小块确认用 TCP 或二进制 ACK 包。

三.实验过程及代码

对于任务一：实现星际通信，我们这里使用的原理是

Delay-Tolerant Networking（DTN）采用"存储—携带—转发"机制， 可跨越超高时延与频繁中断 Tutorials Point。

Bundle Protocol（RFC 5050/草案）在 DTN 之上，将应用数据封装为 bundle，附加生命周期、优先级与路由信息，支持中间节点持久化存储与托管转移（custody transfer）IETF Datatracker。

这里注意，因为传统 TCP 拥塞控制对分钟级以上 RTT 敏感，会导致极端退避，不适合星际场景。

我们可以尝试着写出对此简易的逻辑代码：

```python
from pyd3tn import BPV7, CLA_TCP
bp = BPV7(convergence_layer=CLA_TCP(port=4556))
# 发送 bundle
bundle = bp.create_bundle(
    dest_eid="dtn://mars/endpoint",
    payload=b"Hello from Earth",
    lifetime=3600*24
)
bp.send(bundle)
# 接收 bundle
for received in bp.receive():
    print(received.payload)
```

其中 CLA_TCP 将 bundle 封装在 TCP 连接上传输，中间可切换到 LTP（Licklider Transmission Protocol）等。

对于较为具体的实现，我们可以在此基础上进行功能的追加：对于发送方（我们假设为地球），我们设计了如下发送方代码：

```python
import json, time, socket
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64

# ----------------- 配置 -----------------
DEST_ADDR = ('localhost', 5000)  //这里我们假设的模拟火星节点地址
CUSTODY_TRANSFER = True
LIFETIME = 3600
AES_KEY = b'Sixteen byte key'  //16字节 AES 密钥
WHITE_LIST = ['dtn://mars/rover']  // 目标白名单
# ----------------------------------------

def encrypt_payload(data: bytes) -> bytes:
    cipher = AES.new(AES_KEY, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    packet = {
        'nonce': base64.b64encode(cipher.nonce).decode(),
        'tag': base64.b64encode(tag).decode(),
        'data': base64.b64encode(ciphertext).decode(),
    }
    return json.dumps(packet).encode()

def create_bundle(destination_eid, payload: bytes):
    if destination_eid not in WHITE_LIST:
        raise PermissionError("Destination not authorized for interstellar comms.")

    bundle = {
        'version': 7,
        'destination': destination_eid,
        'creation_time': time.time(),
        'lifetime': LIFETIME,
        'custody_transfer': CUSTODY_TRANSFER,
        'payload': base64.b64encode(encrypt_payload(payload)).decode(),
    }
    return json.dumps(bundle).encode()
```

```
def send_bundle():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(DEST_ADDR)

    data = b"Hello from Earth! Secure message to Mars."
    bundle = create_bundle('dtn://mars/rover', data)

    sock.sendall(bundle)
    print("Bundle sent.")
    sock.close()

if __name__ == "__main__":
    send_bundle()
```

其中引用部分的内容为：

json：将 Python 对象（字典）序列化为 JSON 字符串，方便通过 socket 发送；反之亦然。

time：获取和比较时间戳，用于包生命周期的检查。

socket：socket 标准库，用于建立 TCP 连接。

Crypto.Cipher.AES（来自 PyCryptodome）：提供 AES-GCM 加密/解密；这里是出于对传输的安全性的考虑。base64：将二进制数据（如密文、随机数、标签）编码为可嵌入 JSON 的字符串。

在之后 def encrypt_payload 的内容中，我进行了加密与完整性的设计：

```
def encrypt_payload(data: bytes) -> bytes:
    cipher = AES.new(AES_KEY, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    packet = {
        'nonce': base64.b64encode(cipher.nonce).decode(),
        'tag':   base64.b64encode(tag).decode(),
        'data':  base64.b64encode(ciphertext).decode(),
    }
    return json.dumps(packet).encode()
```

这里的 AES-GCM 模式同时提供保密性（confidentiality）和完整性校验

（integrity/authenticity）。cipher.nonce：每次加密自动生成的随机数，必须随密文一同发送。tag：认证标签，用于解密时验证数据未被篡改。

全部三部分通过 Base64 编码后，打包进一个 JSON 对象，再 .encode() 转为 bytes，以便在更高层打包。

再然后，创建 bunble：

```
def create_bundle(destination_eid, payload: bytes):
    if destination_eid not in WHITE_LIST:
        raise PermissionError("Destination not authorized for interstellar comms.")

    bundle = {
        'version': 7,
        'destination': destination_eid,
        'creation_time': time.time(),
        'lifetime': LIFETIME,
        'custody_transfer': CUSTODY_TRANSFER,
        'payload': base64.b64encode(encrypt_payload(payload)).decode(),
    }
    return json.dumps(bundle).encode()
```

这里我设计的 bunble 各部分内容如下：

version：协议版本号（模拟 BPv7）。

destination：目的地 EID（Endpoint ID）。

creation_time：Unix 时间戳，标记包创建时刻。

lifetime：寿命（秒），超时后接收端丢弃。

custody_transfer：是否需要接收端确认“托管”责任。

payload：先经过 encrypt_payload 返回的 JSON bytes，再做 Base64 编码字符串。

首先，我们检查 destination_eid 是否在 WHITE_LIST 中，否则拒绝发送。最后的打包阶段，我们整个字典序列化为 JSON，再编码为 bytes，准备通过 TCP 发送。

最后是发送方的核心功能，发送信息：

```
def send_bundle():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(DEST_ADDR)

    data = b"Hello from Earth! Secure message to Mars."
    bundle = create_bundle('dtn://mars/rover', data)

    sock.sendall(bundle)
    print("Bundle sent.")
    sock.close()
```

这里，我们首先建立 TCP 连接，然后生成应用数据（示例中是固定的字节串）。

再之后进行打包和加密：调用 create_bundle 得到最终可传输的 bytes。最后将其发送：sendall 保证将所有字节发出；之后关闭连接。

接下来，我们设计了接收方的程序：

```python
import json, socket, base64
from Crypto.Cipher import AES

AES_KEY = b'Sixteen byte key'

def decrypt_payload(encrypted_packet: bytes):
    packet = json.loads(encrypted_packet.decode())
    nonce = base64.b64decode(packet['nonce'])
    tag = base64.b64decode(packet['tag'])
    ciphertext = base64.b64decode(packet['data'])

    cipher = AES.new(AES_KEY, AES.MODE_GCM, nonce=nonce)
    return cipher.decrypt_and_verify(ciphertext, tag)

def handle_bundle(bundle_bytes):
    bundle = json.loads(bundle_bytes.decode())
    expire = bundle['creation_time'] + bundle['lifetime']
    if time.time() > expire:
        print("Bundle expired.")
        return

    encrypted_payload = base64.b64decode(bundle['payload'])
    try:
        payload = decrypt_payload(encrypted_payload)
        print(f"Received message from {bundle['destination']}:\n{payload.decode()}")
        if bundle.get('custody_transfer'):
            print("Custody accepted.")
    except Exception as e:
        print("Integrity check failed:", e)

def run_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('localhost', 5000))
    sock.listen(1)
    print("Listening for bundles...")

    while True:
        conn, _ = conn = sock.accept()
        bundle = conn.recv(4096)
        handle_bundle(bundle)
        conn.close()

if __name__ == "__main__":
    run_server()
```

我们来观察这段代码，首先，引用部分和发送方相对应。接下来的功能模块：

```python
def decrypt_payload(encrypted_packet: bytes):
    packet = json.loads(encrypted_packet.decode())
    nonce = base64.b64decode(packet['nonce'])
    tag    = base64.b64decode(packet['tag'])
    ciphertext = base64.b64decode(packet['data'])


    cipher = AES.new(AES_KEY, AES.MODE_GCM, nonce=nonce)
    return cipher.decrypt_and_verify(ciphertext, tag)
```

这是接收端的功能核心，解析出 nonce、tag、ciphertext，然后用相同的 AES-

GCM 参数执行解密并验证完整性。若 tag 不匹配，会抛出异常，触发“完整性校验失败”。

接下来，我设计了处理与审查：

```
def handle_bundle(bundle_bytes):
    bundle = json.loads(bundle_bytes.decode())
    expire = bundle['creation_time'] + bundle['lifetime']
    if time.time() > expire:
        print("Bundle expired.")
        return

    encrypted_payload = base64.b64decode(bundle['payload'])
    try:
        payload = decrypt_payload(encrypted_payload)
        print(f"Received message from {bundle['destination']}:\n{payload.decode()}")
        if bundle.get('custody_transfer'):
            print("Custody accepted.")
    except Exception as e:
        print("Integrity check failed:", e)
```

这段代码中，首先进行了生命期检查，即如果当前时间 > creation_time + lifetime 那么我们就要丢弃。

然后我们做了解包和解密，这里我们先 Base64 解码，再调用 decrypt_payload。

再之后是异常处理，如果完整性校验失败时捕获异常并报警。

最后是托管转移确认：如果 custody_transfer=True，在此处可触发回执逻辑（示例仅打印确认）。

然后是服务监听端口模块：

```
def run_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('localhost', 5000))
    sock.listen(1)
    print("Listening for bundles...")

    while True:
        conn, _ = sock.accept()
        bundle = conn.recv(4096)
        handle_bundle(bundle)
        conn.close()
```

这里我们绑定并且监听在端口 5000（发送方设置的端口）等待地球端发来的 Bundle。

对于循环接受，我们每次连接接收一个 bundle（简化为一次 recv(4096)），并调用 handle_bundle （即安全审查）处理。

接下来是任务二：设计一套适用于洲际光纤链路的协议集。前提是我们已经完全控制了传输层、网络层和链路层。

对于这个任务，我们首先思考优化的方式：第一个是超高带宽的利用，光纤本身支持数十 Tbps，总目标是在千兆/万兆以上链路上尽可能饱满地发送数据。

第二个内容是高带宽—时延产品（BDP）：单程时延≈ 65 ms，RTT≈ 130 ms，BDP（带宽×RTT）非常大，需要足够大的窗口和足够快的拥塞控制算法。

第三个内容是，极低误码率，但偶发丢包：物理误码率极低，但再生中继、设备切换可能引入短暂丢包或抖动。

第四个内容是部署可行性：需对现有 IP/TCP 栈做最小改动，并能在链路中间分阶段部署优化。第五个内容就是透明性：客户端和服务器无需修改，仅在链路两端或中间插入优化层即可提升性能。

有了这些想法之后，我们尝试进行分层设计：

首先，对于物理层我们由于波长复用、高密度的问题，我们使用 DWDM 多波长通道；启动 RS-LDPC FEC；OSNR 优化

然后，对于链路层，由于帧头开销较大，我们就可以选择 Jumbo Frame (MTU=9000)。对于网络层，由于 IP 分片/重组开销，我们选择了 IPv6（更少中间分片）或者其实 MPLS/MPLS-TP 我觉得也是可以的。对于传输层，因为有高 BDP，并且环境拥塞控制欠佳，所以我们选择 TCP BBR + Window Scaling。

分层设计完成之后，我们再考虑，代理加速（PEP），因为单条 TCP 难以快速吃满管道，所以我们可以将协议设计成并行多路拆连接 + 序号重排序。

而对于安全而言，由于数据保密、完整性、认证，我们可以使用 TLS/DTLS 隧道，或者在 PEP 应用之间加上 TLS。

完成底层的设计之后，我们开始考虑具体的实现：

首先。系统层面优化

对于 Jumbo Frame（MTU 9000）：增大发送单元，减小协议头相对开销，让每个 Ethernet 帧承载更多数据，提高大数据传输效率。

```
oslab@oslab-virtual-machine:~/桌面$ sudo ip link set eth0 mtu 9000
[sudo] oslab 的密码：
Cannot find device "eth0"
```

TCP BBR 拥塞控制：传统 TCP（CUBIC/NewReno）在高带宽-时延（BDP）场景下，窗口增长太慢且易有队头阻塞；BBR 直接根据实时测得的带宽与 RTT 构造发送速率，既填满管道又不过载。

```
oslab@oslab-virtual-machine:~/桌面$ sudo tee /etc/sysctl.d/99-bbr.conf << 'EOF'
net.core.default_qdisc = fq
net.ipv4.tcp_congestion_control = bbr
EOF
sudo sysctl --system
net.core.default_qdisc = fq
net.ipv4.tcp_congestion_control = bbr
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.default.use_tempaddr = 2
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
kernel.kptr_restrict = 1
* Applying /etc/sysctl.d/10-magic-sysrq.conf ...
kernel.sysrq = 176
* Applying /etc/sysctl.d/10-network-security.conf ...
net.ipv4.conf.default.rp_filter = 2
net.ipv4.conf.all.rp_filter = 2
* Applying /etc/sysctl.d/10-ptrace.conf ...
```

Socket 缓冲区调优：在高 BDP 链路上，默认 TCP 缓冲区（约 64 KB）可能不够，需扩到 tens of MB 级别，以匹配链路延迟所需的“管道大小”。

```
oslab@oslab-virtual-machine:~/桌面$ sudo sysctl -w net.core.rmem_max=33554432
sudo sysctl -w net.core.wmem_max=33554432
sudo sysctl -w net.ipv4.tcp_rmem="4096 87380 33554432"
sudo sysctl -w net.ipv4.tcp_wmem="4096 65536 33554432"
net.core.rmem_max = 33554432
net.core.wmem_max = 33554432
net.ipv4.tcp_rmem = 4096 87380 33554432
net.ipv4.tcp_wmem = 4096 65536 33554432
```

对于 PEP 代理原理与代码解析，通过网上查阅到的资料显示，星—港美链路 RTT ≈ 130 ms，单一 TCP 连接的拥塞窗口在丢包或慢启动时要数秒才能“吃饱”这条管道。

PEP 的思路是：

拆连接：在链路入口（香港端）和出口（洛杉矶端）各放一个代理。并行多流：将客户端发来的数据按包编号轮询分配给多个（如 4）并行 TCP 会话，每条子流各自拥塞控制，互不干扰。重排序合并：在出口再按包编号顺序还原，再写入到真实服务器连接中。

这样，即使单条子流遇到丢包退避，其他子流仍可继续发送，整体吞吐更平稳且更高。

基于此，我们设计香港端的程序：

```
N_STREAMS = 4
LA_PEER = ('la.pep.example.com', 7000)
LOCAL_BIND = ('0.0.0.0', 8000)

streams = []
for i in range(N_STREAMS):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
    s.connect(LA_PEER)
    streams.append(s)

def handle_client(conn):
    seq = 0
    while True:
        data = conn.recv(16384)
        if not data: break
        header = struct.pack('!IH', seq, len(data))
        sock = streams[seq % N_STREAMS]
        sock.sendall(header + data)
        seq += 1
    conn.close()
```

其中 N_STREAMS 表示并行子流数目，通常选 4–8，根据链路丢包/RTT 特性调优。而 streams 则表示持久 TCP 套接字列表，一次性建立好，不随客户端连接重建，减少握手延迟。

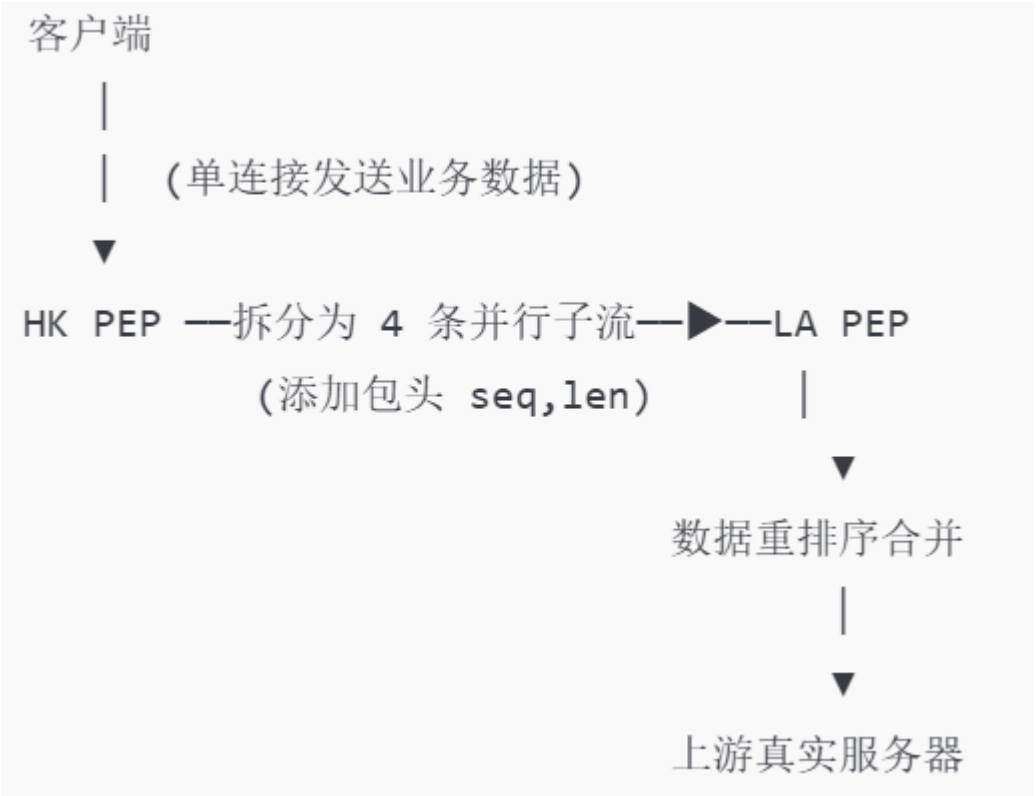
对于接收端，即洛杉矶：

```
HK_BIND = ('0.0.0.0', 7000)
UPSTREAM_SERVER = ('10.0.0.100', 9000)

upstream = socket.socket()
upstream.connect(UPSTREAM_SERVER)
def worker(stream_sock):
    buffer = {}
    expected_seq = 0
    while True:
        hdr = stream_sock.recv(6)
        if len(hdr) < 6: break
        seq, length = struct.unpack('!IH', hdr)
        data = b''
        while len(data) < length:
            data += stream_sock.recv(length - len(data))
        buffer[seq] = data
        while expected_seq in buffer:
            upstream.sendall(buffer.pop(expected_seq))
            expected_seq += 1
    stream_sock.close()


listener = socket.socket()
listener.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
listener.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
listener.bind(HK_BIND)
listener.listen(N_STREAMS)
```

UPSTREAM_SERVER 表示后端真正处理业务的服务器地址。Upstream 表示一条单一 TCP 链接，用于最终数据汇总后发送。buffer：使用字典来暂存乱序到达的包。expected_seq：当前等待的最低序号，确保严格按序提交上游。

这里使用到的合并策略：一旦 buffer 中包含 expected_seq，就立刻发送并 expected_seq+=1，直至下一个缺失包。

对于该协议集，我们可以梳理一下其流程：



最后是第三个任务：设计校园网传输大文件传输程序，这里我选择了使用基于 UDP 的稳定传输协议来设计该程序，程序传输的核心思路是：

分包与编号：将大文件切分成固定大小（如 8 KB）的数据包，并为每个包分配唯一的序号。滑动窗口：发送端维护可发送但尚未确认的包的窗口，接收端通过 ACK 通知收到的最高连续序号；重传与超时：发送端为每个未确认的数据包启动定时器，超时后重传；并发与流控：可配置窗口大小（默认 32），根据网络丢包率和带宽动态调整。

程序暂定的流程为：首先连接建立，由客户端向服务器 UDP 端口发送 “START” 请求包，包含欲接收的文件名。

然后，服务器返回同意并初始化传输。

针对数据传输，我们让服务器将文件分为 N 个大小为 PAYLOAD_SIZE（如 8 KB）的块，每块加上 4 字节序号 (uint32)。然后维护一个滑动窗口 W（默认 32 包），

可同时发送窗口内所有未确认的数据包。接着，每发送一个包即启动定时器；收到对应 ACK（服务端收到包含序号的确认包）后清除此包并滑动窗口。

对于重传机制，我们指定，如果包的定时器超时（默认 0.5 s）未收到 ACK，则重传该包。当收到重复 ACK 或 NACK 时，可快速重传对应包。

最后，传输结束，发送完所有包后，服务器发送特殊 FIN 包。客户端收到 FIN 后回复 FIN-ACK 并关闭。

基于以上思路，我们开始编写程序代码：

发送端代码如下：

```
#!/usr/bin/env python3
import socket, threading, struct, time, os

SERVER_PORT = 9000
PAYLOAD_SIZE = 8192
WINDOW_SIZE = 8
TIMEOUT = 0.5  # seconds

class Sender:
    def __init__(self, filename, client_addr, sock):
        self.filename = filename
        self.client = client_addr
        self.sock = sock
        self.file_size = os.path.getsize(filename)
        self.total_pkts = (self.file_size + PAYLOAD_SIZE - 1) // PAYLOAD_SIZE
        self.base = 0
        self.next_seq = 0
        self.lock = threading.Lock()
        self.timers = {}  # seq -> timer threading.Timer

  def start(self):
      # 向客户端确认
      self.sock.sendto(struct.pack('!I', self.total_pkts), self.client)
      # 启动 ACK 监听线程
      threading.Thread(target=self.ack_listener, daemon=True).start()
      with open(self.filename, 'rb') as f:
          while self.base < self.total_pkts:
              self.lock.acquire()
              while self.next_seq < self.base + WINDOW_SIZE and self.next_seq < self.total_pkts:
                  f.seek(self.next_seq * PAYLOAD_SIZE)
                  data = f.read(PAYLOAD_SIZE)
                  pkt = struct.pack('!I', self.next_seq) + data
                  self.sock.sendto(pkt, self.client)
                  # 启动定时重传
                  timer = threading.Timer(TIMEOUT, self.retransmit, args=(self.next_seq,))
                  self.timers[self.next_seq] = timer
                  timer.start()
                  self.next_seq += 1
              self.lock.release()
              time.sleep(0.01)
      # 发送 FIN
      self.sock.sendto(b'FIN', self.client)
```

```
    def retransmit(self, seq):
        with self.lock:
            if seq >= self.base:
                f = open(self.filename, 'rb')
                f.seek(seq * PAYLOAD_SIZE)
                data = f.read(PAYLOAD_SIZE)
                pkt = struct.pack('!I', seq) + data
                self.sock.sendto(pkt, self.client)
                # 重置定时器
                t = threading.Timer(TIMEOUT, self.retransmit, args=(seq,))
                self.timers[seq] = t
                t.start()
                f.close()

    def ack_listener(self):
        while True:
            msg, _ = self.sock.recvfrom(8)
            if msg.startswith(b'ACK'):
                ack_seq = struct.unpack('!I', msg[3:])[0]
                with self.lock:
                    if ack_seq in self.timers:
                        self.timers[ack_seq].cancel()
                        del self.timers[ack_seq]
                    if ack_seq == self.base:
                        # 向前滑动
                        while self.base not in self.timers and self.base < self.total_pkts:
                            self.base += 1
            elif msg == b'FIN-ACK':
                break
```

```
def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', SERVER_PORT))
    print(f"Server listening on UDP port {SERVER_PORT}")
    while True:
        data, addr = sock.recvfrom(1024)
        if data.startswith(b'START'):
            filename = data[5:].decode()
            if not os.path.exists(filename):
                sock.sendto(b'ERR', addr)
                continue
            sender = Sender(filename, addr, sock)
            threading.Thread(target=sender.start, daemon=True).start()

if __name__ == "__main__":
    main()
```

接收端代码如下：

```python
#!/usr/bin/env python3
import socket, struct, threading, time

SERVER_IP = '192.168.219.132'
SERVER_PORT = 9000
PAYLOAD_SIZE = 8192

def receive_file(save_name, request_name):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.settimeout(5)
    sock.sendto(b'START' + request_name.encode(), (SERVER_IP, SERVER_PORT))

    # 接收总包数
    data, _ = sock.recvfrom(4)
    total_pkts = struct.unpack('!I', data)[0]
    buffer = [None] * total_pkts
    recv_count = 0
    expected = 0

    start_time = time.time()
    while recv_count < total_pkts:
        try:
            pkt, _ = sock.recvfrom(PAYLOAD_SIZE + 4)
        except socket.timeout:
            continue
        if pkt == b'FIN':
            break
        seq = struct.unpack('!I', pkt[:4])[0]
        payload = pkt[4:]
        if buffer[seq] is None:
            buffer[seq] = payload
            recv_count += 1
        # 发送 ACK
        sock.sendto(b'ACK' + struct.pack('!I', seq), (SERVER_IP, SERVER_PORT))
    # 发送 FIN-ACK
    sock.sendto(b'FIN-ACK', (SERVER_IP, SERVER_PORT))

    # 写文件
    with open(save_name, 'wb') as f:
        for chunk in buffer:
            f.write(chunk)
    elapsed = time.time() - start_time
    print(f"Received {total_pkts * PAYLOAD_SIZE:,} bytes in {elapsed:.2f} s, throughput = {total_pkts*PAYLOAD_SIZE/elapsed/1024/1024:.2f} MB/s")

if __name__ == "__main__":
    receive_file('downloaded.bin', 'largefile.bin')
```

我 们 运 行 两 个 程 序 来 测 试 其 是 否 能 正 确 传 输 文 件 ：

```
oslab@oslab-virtual-machine:~/桌面/sy$ dd if=/dev/urandom of=largefile.bin bs=1M
 count=1024
记录了1024+0 的读入
记录了1024+0 的写出
1073741824字节（1.1 GB，1.0 GiB）已复制，4.65638 s，231 MB/s
```

首先我们随机生成一个 1G 左右大小的文件。然后运行程序。

```
oslab@oslab-virtual-machine:~/桌面/sy$ python3 sr.py
Server listening on UDP port 9000
Exception in thread Thread-1 (start):
Traceback (most recent call last):
  File "/usr/lib/python3.10/threading.py", line 1016, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.10/threading.py", line 953, in run
    self._target(*self._args, **self._kwargs)
  File "/home/oslab/桌面/sy/sr.py", line 37, in start
    timer.start()
  File "/usr/lib/python3.10/threading.py", line 935, in start
    _start_new_thread(self._bootstrap, ())
RuntimeError: can't start new thread
```

发现发送端出现了该错误，即无法创建新的线程，应该是由于 linux 中的轻量线

程资源耗尽了，无法再创建，于是我试着把窗口大小由 32 改到了 8，再试一次，发现还是会有同样的错误，所以我们只能修改代码，修改的思路为用单线程定时轮询取代大量 threading.Timer。

```
        while not self.stop_event.is_set():
            now = time.time()
            to_retx = []
            with self.lock:
                for seq, ts in self.sent_times.items():
                    # 超时且仍在滑动窗口之内
                    if now - ts >= TIMEOUT and seq >= self.base:
                        to_retx.append(seq)

            # 在锁外逐个重传
            for seq in to_retx:
                with open(self.filename, 'rb') as f:
                    f.seek(seq * PAYLOAD_SIZE)
                    data = f.read(PAYLOAD_SIZE)
                    pkt = struct.pack('!I', seq) + data
                    self.sock.sendto(pkt, self.client)
                # 更新重传时间戳
                with self.lock:
                    self.sent_times[seq] = now

            time.sleep(CHECK_INTERVAL)
```

代码修改部分

重新运行代码

```
oslab@oslab-virtual-machine:~/桌面/sy$ python3 sr.py
Server listening on UDP port 9000
```

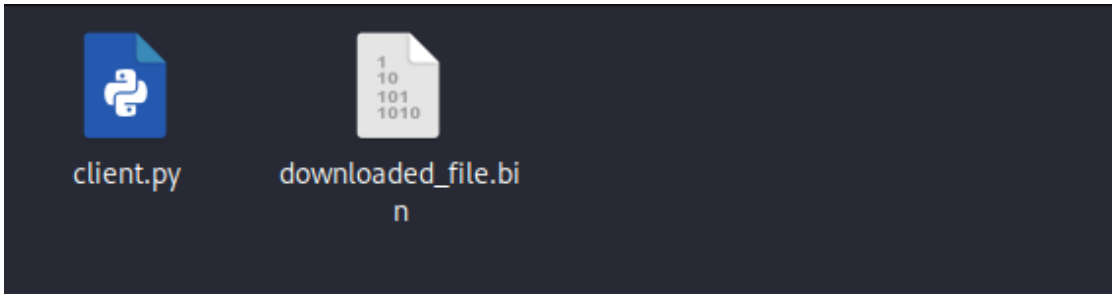
没有报错

```
┌──(kali㉿kali)-[~/桌面/sy8]
└─$ python3 client.py
Expecting 131072 packets...
Received 500/131072 packets (0.38%)
Received 1000/131072 packets (0.76%)
Received 1500/131072 packets (1.14%)
Received 2000/131072 packets (1.53%)
Received 2500/131072 packets (1.91%)
Received 3000/131072 packets (2.29%)
Received 3500/131072 packets (2.67%)
Received 4000/131072 packets (3.05%)
Received 4500/131072 packets (3.43%)
Received 5000/131072 packets (3.81%)
Received 5500/131072 packets (4.20%)
Received 6000/131072 packets (4.58%)
Received 6500/131072 packets (4.96%)
Received 7000/131072 packets (5.34%)
Received 7500/131072 packets (5.72%)
Received 8000/131072 packets (6.10%)
Received 8500/131072 packets (6.48%)
Received 9000/131072 packets (6.87%)
Received 9500/131072 packets (7.25%)
Received 10000/131072 packets (7.63%)
Received 10500/131072 packets (8.01%)
Received 11000/131072 packets (8.39%)
Received 11500/131072 packets (8.77%)
Received 12000/131072 packets (9.16%)
Received 12500/131072 packets (9.54%)
Received 13000/131072 packets (9.92%)
Received 13500/131072 packets (10.30%)
Received 14000/131072 packets (10.68%)
Received 14500/131072 packets (11.06%)
Received 15000/131072 packets (11.44%)
```

可以看到接收端正在接收该文件。接收完毕之后，我们看到：

```
eived 129000/131072 packets (98.42%)
eived 129500/131072 packets (98.80%)
eived 130000/131072 packets (99.18%)
eived 130500/131072 packets (99.56%)
eived FIN from server.
 Warning: 78 packets were lost. The file may be incomplete.
Received 1,073,741,824 bytes in 24.04s, throughput = 42.59 MB/s
```

发现其中有 78 个 package 丢失了，同时发现在目录下找到了我们的接收文件：



四.结论

这次的第八次作业，可以说是一次非常全面、很“实战”的网络通信设计与实现训练。无论是从理论角度出发的星际通信协议构思，还是针对实际应用的校园 WiFi 下的大文件传输实验，都充满了技术挑战，也带来了很多意想不到的收获。

首先是星际通信部分。这部分真的挺有意思，毕竟平时我们谈到网络传输，最多也就是“跨国”、“跨洋”，很少有人会认真去考虑“跨星球”的问题。而这次，我们尝试从 DTN（Delay/Disruption-Tolerant Networking） 出发，思考如何在动辄几十分钟延迟、动不动断联的深空环境中实现可靠通信。

我们借助了 Bundle Protocol（BP） 来打包数据、实现“存储—携带—转发”，还加入了托管传输（custody transfer）和生命周期管理，并基于 AES-GCM 实现了加密与完整性保护。虽然只是简化版的模拟实现，但在动手写代码的过程中，我真切地体会到深空通信的复杂性——那是一种“只发得起一次，必须发得准”的感觉。设计的时候，我们必须考虑到每一个潜在失败点。

相比之下，洲际光纤链路的协议设计虽然“接地气”多了，但挑战一点也不少。在拥有超大带宽的前提下，我们要面临的，其实是如何充分吃满这根“管道”。

这部分我们从物理层、链路层、网络层到传输层，一层一层去优化，比如 DWDM、Jumbo Frame、TCP BBR、PEP 加速器等都搬上了台面。最有意思的部分是 PEP 设计——通过多条 TCP 子流并行传输，然后在接收端重排合并。它就像是一种“明明可以靠速度，但我偏要靠分身”的解决方案，用结构优化来对抗带宽-时延产品高带来的瓶颈。

而通过简单实现香港端和洛杉矶端的两个代理程序，我们能直观地看到这种“多流 + 重组”的威力：更高吞吐、更抗抖动，也更适合高质量、大体积数据的传输场

景。

到了最后一个实验，直接落地到我们自己的笔记本之间。这个实验可以说是“最能感同身受”的部分了。

这次我们选择了基于 UDP 的可靠传输协议实现，自己动手实现了分片、ACK、窗口、重传、超时、滑动窗口等等。这个过程一开始看上去似乎不太难，真正写起来才发现问题很多，比如：

threading.Timer 会耗尽系统线程资源（尤其在 Linux 上）；丢包之后怎么高效重传；窗口大小该怎么选；超时时间长了慢、短了重传频繁；程序本身对系统资源的占用也可能成为瓶颈。

但也正是这些“坑”，让我们在调试过程中不断磨合出更合理的策略——最终选择用单线程+循环定时器替代大量线程、调整窗口大小，终于稳定完成了一个 1GB 文件的传输。

参考文献

1. https://www.nasa.gov/wp-content/uploads/2023/09/dtn-tutorial-v3.2-0.pdf. Delay- and Disruption-Tolerant Networks (DTNs). Forrest Warthman, Warthman Associates. 2015.9.14.

[2]https://www.techtarget.com/searchnetworking/definition/delay-tolerant-network. delay-tolerant network (DTN). Rahul Awati.2024.11.14.

[3] https://wiki.python.org/moin/UdpCommunication UDP Communication

. Python Software Foundation.2021.01.24.

[4] https://stackoverflow.com/questions/59255054/python-udp-socket-client-recevfrom-is-very-slow. python UDP socket client recevfrom is very slow. Sam Mason.2023.8.15.

[5] https://www.tutorialspoint.com/what-is-the-maximum-number-of-threads-per-process-in-linux. Maximum Number of Threads per Process in Linux. Mukul Latiyan.2022.3.24

[6]https://forum.greenbone.net/t/runtimeerror-cant-start-new-threa-eption-in-thread/16789. RuntimeError: can't start new thread / Exception in thread. PBSH.2024.9.16.

[7] https://askubuntu.com/questions/95022/what-are-the-main-differences-between-virtualbox-networking-type. What are the main differences between VirtualBox networking types?. jrg. 2022.8.15.