《计算机网络第七次作业》

班级：信安 2302 班

学号：202308060227

姓名：石云博

目录

一.问题描述

1.设计一个支持可靠传输的一对一数据包通信协议，并强烈要求发送方和接收方都会在某个时候知道每个数据包的最终命运，无论是接收还是丢失。

2.尽可能提高协议的效率，减少其最大延迟、平均延迟、增加其最大吞吐量、平均吞吐量或您认为重要的任何属性。针对各种网络条件（如随机错误、 突发错误等）分析并证明您的设计。

3.使用 UDP 实现协议，在宿舍网络环境中测量其最大吞吐量。有什么方法可以进一步提升性能吗?

4.尝试为 1 中的要求设计一到多的广播协议

二.问题分析

首先，想要进行一对一数据包可靠传输协议的设计，我们首先需要知道协议设计目标 1 可靠性保证：无论网络环境如何，最终双方都能知道每个数据包是否被成功传输或因故障丢失。2 确认机制：使用正向确认（ACK）与负向确认（NAK），配合超时重传机制。3 拥塞与流量控制：采用滑动窗口机制，不仅能降低延迟，还能提高吞吐量。

其中的协议基本机制中，数据包编号及校验有：为每个数据包分配唯一序号，同时附带校验和（CRC 或其他机制）用以检测传输错误。以及这种编号保证双方能准确记录已发送和已确认的数据。

然后，想要实现正向确认（ACK）与负向确认（NAK），我们需要知道

ACK 机制：当接收方成功校验并接收数据包后，立即发送 ACK（可以是累积确认，确认所有顺序内的数据包）。

NAK 机制：若检测到错误或者丢失（例如，收到的序列不连续），接收方可发送 NAK 提醒发送方重传缺失数据。

此外，设计要求双方最终都知道结果，所以每个数据包的 ACK/NAK 都必须经过“最终确认”阶段（例如，双方在一段时间内没有出现后续的疑问后，进入“决断状态”）。

然后，对于超时重传与状态清理

发送方为每个发送出去的数据包启动超时计时器。如果在预设时间内未收到 ACK，则发起重传；重传次数可以设定上限，超过上限后，认为数据包不可达并将状态标记为“失败”。

为确保双方最终一致，接收方也在经过一定的等待（例如等待可能的重传数据包）

后，将最终状态反馈给发送方。

最终状态报告可以采用双方约定的“决断消息”，当双方都已确认无后续重传后，则该数据包的命运确定。

想要实现滑动窗口机制，我们得进行相应设计：

实现流水线式传输：允许发送窗口中有多个未确认数据包，同时接收方按序确认。

针对大吞吐量要求，窗口大小应根据信道带宽和往返延迟（RTT）进行调整：

$$窗口大小 \approx 带宽 \times RTT$$

同时，需要设计窗口中的数据包应支持乱序重排和选择性确认（Selective Acknowledgment, SACK），以应对部分数据包丢失的情况。

最后，针对双方最终命运知道的策略

1 确认累计与决断消息：接收方可以周期性地将已收到的数据包序号段汇总发送出去；发送方在接收到一定期间内所有数据包序号的汇总后，若发现缺失则启动补传机制。如果在一个决策周期内所有数据包均已成功应答，则双方进入“传输完成”状态。

2 状态同步轮询：为了确保最终双方知道状态，也可设计“状态同步”阶段，即在所有数据传输结束后，发送双方交换最后一个状态确认消息，确保双方结果一致。

三.实验过程及代码

首先，我们要确定协议设计的实现，实现一个基于 UDP 的可靠传输协议，关键在于在应用层补充以下机制：

数据包编号与校验

每个数据包带有唯一的序号（Sequence Number），以及可选的校验和（CRC 或

MD5）以验证数据完整性。

确认与反馈（ACK/NAK）

接收方在正确接收到数据后，立即返回确认消息。对于错误数据或缺失的数据，可以选择发送负确认（NAK）或者采用选择性确认（SACK）的方法。

超时重传机制

发送方为每个数据包启动计时器，如果在超时时间内未收到 ACK，则进行重传。重传次数可有限制，达到上限后认为该数据包丢失，状态终结。

滑动窗口和流水线

为提高吞吐量，采用滑动窗口技术，即允许同时发送多个数据包并独立确认；收到乱序的数据包时，依靠序号进行重排序。

最终确定机制

在传输终止后，双方交换状态（例如“传输完毕”消息），保证发送方和接收方都能最终知道每个数据包是被正确接收还是认定丢失。

然后，我们可以尝试着设计传输协议中的数据包格式与数据结构。参考 TCP 协议的报文格式，我们可以将数据包格式设计为：

头部字段（总长度固定，例如 20 字节，可自定义）：

版本号（1 字节）：协议版本号

标志位（1 字节）：指示数据包类型（数据、ACK、NAK、控制信令等）

序号 (Seq)（4 字节）：唯一递增序号，用来排序和确认

确认号 (Ack)（4 字节）：用于确认对方收到的最后一个连续数据包

窗口大小（4 字节）：当前发送方或接收方的可用窗口（可选）

校验和（4 字节）：对头部和有效载荷的校验码

有效负载：实际携带的数据内容

在实现中，可以使用结构体或类来描述数据包（例如在 C/C++中用 struct，在 Python 中可以用字节数组结合 struct 模块）。

```python
import struct

# 定义头部格式："!BBIII" 代表：
# - B: 版本号 (1字节)
# - B: 标志位 (1字节)
# - I: 序号 (4字节)
# - I: 确认号 (4字节)
# - I: 校验和 (4字节)

HEADER_FORMAT = "!BBIII"
HEADER_SIZE = struct.calcsize(HEADER_FORMAT)

def pack_header(version, flags, seq, ack, checksum):
    return struct.pack(HEADER_FORMAT, version, flags, seq, ack, checksum)

def unpack_header(data):
    return struct.unpack(HEADER_FORMAT, data[:HEADER_SIZE])
```

想要实现具体的发送流程，我们还需要知道，数据切分：

若应用层数据较大，则在发送前将数据切分成不超过 MTU 大小的块（保持头部占用后，通常数据负载小于 1500- (IP 头+UDP 头+自定义头)）。

缓冲与滑动窗口：

定义发送缓冲区和待确认队列。缓冲区存放已发送但未确认的数据包。窗口大小可根据网络 RTT 和带宽动态调整。

数据包发送：

对于缓冲区中每个数据包：将数据包打包为二进制字节串；利用 UDP socket 发送出去；启动计时器，并在本地记录发送时间等状态。

计时器监控重传：

检查未确认的数据包，如果超过预设重传超时时间（例如 RTT 的 1.5～2 倍），则触发重传。如果重传次数超过最大阈值，则标记数据包丢失。

进行数据接收的时候，使用 UDP socket 非阻塞或异步模式接收数据包，得到数据后解析头部和负载。对于校验和验证，对收到数据进行校验，若校验失败，则丢弃数据，并可以选择发送 NAK 反馈。

想要实现序号确认与缓存排序，将收到的数据按序号存入接收缓冲区。如果出现乱序，则暂时缓存，直到缺失数据重传过来。发送 ACK 的时候，一旦连续序号数据达到一定数目，接收方在反馈数据时，可使用“捎带确认”技术：在回复 ACK 时，将下一个期待数据包的序号作为确认号发给对方。可以采用计数器（如接到一定数量或定时触发）发送确认，以减少 ACK 风暴，特别在一对多广播中非常重要。

想要实现状态同步与连接管理：为了确保双方最终得知每个数据包的状态，设计如下机制：连接建立：在开始传输前采用类似“三次握手”的过程来协商会话参数（例如初始序号、窗口大小），即使 UDP 没有连接特性，应用层可以模拟此过程。

心跳检测：

在传输过程中定时发送心跳消息，确保双方活跃。当一方长时间未收到心跳时，视为连接中断，对所有未被确认数据包进行处理。

在最终确认阶段中，在数据全部发送完毕后，双方交换一个“传输完成”消息，确保双方均退出等待状态，最终得知哪些数据包成功，哪些超限判定为丢失。

根据以上的设计分析，我们可以设计出相对应的传输方和接收方的代码：

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

SERVER_IP = "127.0.0.1"
SERVER_PORT = 5005
SERVER_ADDR = (SERVER_IP, SERVER_PORT)

TIMEOUT = 2.0
MAX_RETRIES = 5
WINDOW_SIZE = 5

HEADER_FORMAT = "!I"
HEADER_SIZE = struct.calcsize(HEADER_FORMAT)

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.settimeout(0.5)
```

发送端代码，进行参数配置。

```python
def pack_packet(seq, payload):
    """将序号和有效负载打包成数据包"""
    header = struct.pack(HEADER_FORMAT, seq)
    return header + payload

def unpack_packet(packet):
    """从数据包中解出序号和有效负载"""
    header = packet[:HEADER_SIZE]
    seq = struct.unpack(HEADER_FORMAT, header)[0]
    payload = packet[HEADER_SIZE:]
    return seq, payload

def send_packet(seq, payload):
    packet = pack_packet(seq, payload)
    sock.sendto(packet, SERVER_ADDR)
    with buffer_lock:
        send_buffer[seq] = (packet, time.time(), 0)
    print(f"[SENDER] 发送 seq {seq}, 长度 {len(packet)}")
```

数据包打包与解包函数

```
def sender(data):
    global next_seq, send_finished
    chunks = [data[i:i+100] for i in range(0, len(data), 100)]
    total_chunks = len(chunks)
    print(f"[SENDER] 数据总共有 {total_chunks} 个数据块")

    for chunk in chunks:
        while True:
            with buffer_lock:
                if len(send_buffer) < WINDOW_SIZE:
                    seq = next_seq
                    next_seq += 1
                    break
            time.sleep(0.01)
        send_packet(seq, chunk)
    while True:
        with buffer_lock:
            if not send_buffer:
                break
```

发送函数。

```
    while True:
        with buffer_lock:
            if not send_buffer:
                break
        time.sleep(0.1)
    sock.sendto(b'FIN', SERVER_ADDR)
    send_finished = True
    print("[SENDER] 数据发送完毕，等待 FIN_ACK")
```

等待所有数据发送完毕。

```python
def ack_listener():
    global send_finished
    while True:
        try:
            data, addr = sock.recvfrom(1024)
        except socket.timeout:
            if send_finished:
                break
            continue
        if data == b'FIN_ACK':
            print("[SENDER] 收到 FIN_ACK, 传输结束")
            break
        ack_seq = struct.unpack(HEADER_FORMAT, data)[0]
        with buffer_lock:
            removed = [s for s in send_buffer if s <= ack_seq]
            for s in removed:
                del send_buffer[s]
                print(f"[SENDER] 收到 ACK, 确认 seq {s}")
    print("[SENDER] ACK监听线程退出")
```

ACK 机制确认实现函数。

```python
def retransmission_monitor():
    while True:
        time.sleep(0.1)
        with buffer_lock:
            now = time.time()
            remove_seqs = []
            for seq, (packet, send_time, retries) in list(send_buffer.items()):
                if now - send_time > TIMEOUT:
                    if retries < MAX_RETRIES:
                        sock.sendto(packet, SERVER_ADDR)
                        send_buffer[seq] = (packet, now, retries + 1)
                        print(f"[SENDER] 重传 seq {seq}, 第 {retries + 1} 次")
                    else:
                        print(f"[SENDER] seq {seq} 超过最大重传次数, 放弃")
                        remove_seqs.append(seq)
            for seq in remove_seqs:
                del send_buffer[seq]
        with buffer_lock:
            if send_finished and not send_buffer:
```

重传机制实现函数。

```python
if __name__ == '__main__':
    data = ("这是一段需要可靠传输的数据，经过切分后每个数据块不超过100字节，"
            "发送方采用滑动窗口、ACK确认以及超时重传机制保证所有数据能够被正确接收。"
            "测试完毕后，发送方会发送FIN结束标志。").encode('utf-8')

    ack_thread = threading.Thread(target=ack_listener, daemon=True)
    ack_thread.start()
    retrans_thread = threading.Thread(target=retransmission_monitor, daemon=True)
    retrans_thread.start()
    sender(data)
    ack_thread.join()
    retrans_thread.join()
    sock.close()
    print("[SENDER] 发送端程序退出")
```

主函数。

对于接收方：

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

BIND_IP = "0.0.0.0"
BIND_PORT = 5005
BIND_ADDR = (BIND_IP, BIND_PORT)

HEADER_FORMAT = "!I"
HEADER_SIZE = struct.calcsize(HEADER_FORMAT)

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(BIND_ADDR)
print(f"[RECEIVER] 监听 {BIND_ADDR}")
```

参数配置。

```python
while True:
    packet, addr = sock.recvfrom(2048)
    if packet == b'FIN':
        sock.sendto(b'FIN_ACK', addr)
        print("[RECEIVER] 收到 FIN，发送 FIN_ACK，结束接收")
        break
    seq, payload = struct.unpack(HEADER_FORMAT, packet[:HEADER_SIZE])[0], packet[HEADER_SIZE
    print(f"[RECEIVER] 收到 seq {seq} 数据: {payload.decode('utf-8', errors='replace')}")
    if seq == expected_seq:
        print(f"[RECEIVER] 处理 seq {seq} 数据: {payload.decode('utf-8', errors='replace')}")
        expected_seq += 1

        while expected_seq in received_chunks:
            cached = received_chunks.pop(expected_seq)
```

主循环代码。

```
    ack_value = expected_seq - 1
    ack_packet = struct.pack(HEADER_FORMAT, ack_value)
    sock.sendto(ack_packet, addr)
    print(f"[RECEIVER] 发送 ACK, 确认到 seq {ack_value}")
```

ACK 确认代码。

我们还需要测定接收方和发送方直接能否直接建立连接进行通信，所以我们再建立一对发送方和接收方的确认程序。

```
import socket

def send_message():
    SERVER_IP = '175.10.107.61'
    SERVER_PORT = 5005

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((SERVER_IP, SERVER_PORT))
    s.sendall(b"Hello from client!")
    data = s.recv(1024)
    print("[CLIENT] Received:", data.decode())
    s.close()

send_message()
```

发送方确认代码，server_IP 设置为接收方 IP。如果对方接收并建立起了连接，那么就会收到”Hello from client”。

```python
import socket

def start_server():
    HOST = '175.10.107.61'
    PORT = 5005

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen(1)
    print(f"[SERVER] Listening on {HOST}:{PORT}")

    conn, addr = s.accept()
    print(f"[SERVER] Connection from {addr}")

    while True:
        data = conn.recv(1024)
        if not data:
            break
        print("[SERVER] Received:", data.decode())
        conn.sendall(b"ACK")

    conn.close()
    s.close()

start_server()
```

接收方确认程序代码。

然后我们分别在发送方和接收方运行各自的代码，得到结果：

```
oslab@oslab-virtual-machine:~/桌面$ python3 js1.py
[SERVER] Listening on 175.10.107.61:5005
[SERVER] Connection from ('175.10.104.62', 47788)
[SERVER] Received: Hello from client!
```

可以看到发送方和接受方之间已经成功建立了连接。

然后我们运行正式的发送接收文件：

```
[SENDER] 数据总共有 3 个数据块
[SENDER] 发送 seq 1，长度 104
[SENDER] 发送 seq 2，长度 104
[SENDER] 发送 seq 3，长度 56
[SENDER] 收到 ACK，确认 seq 1
[SENDER] 收到 ACK，确认 seq 2
[SENDER] 收到 ACK，确认 seq 3
[SENDER] 数据发送完毕，等待 FIN_ACK
[SENDER] 收到 FIN_ACK，传输结束
[SENDER] ACK监听线程退出
[SENDER] 重传监控线程退出
[SENDER] 发送端程序退出
```

发送方结果

```
slab@oslab-virtual-machine:~/桌面$ python3 js.py
RECEIVER] 正在监听 ('0.0.0.0', 5005)...
RECEIVER] 收到 seq 1，长度 100
RECEIVER] 收到 seq 2，长度 100
RECEIVER] 收到 seq 3，长度 52
RECEIVER] 收到 FIN，发送 FIN_ACK，准备结束
RECEIVER] 全部数据接收完成：
```

接收方结果。

可以看到正确地实现了传输的要求。其中，每次接收方收到的 seq 长度减小了 4。这是由于数据包的拆包与解析过程中，**头部大小（Header）和负载（Payload）**的处理顺序或者数据处理中的偏差引起的。在接收方解析包时，使用了 HEADER_FORMAT = "!I" 进行头部解析，这意味着包的前 4 个字节（头部）是用来表示 seq 序列号的。所以接收方解析之后其长度会减少 4.证明了该协议的可行性。

第二个任务，我们需要根据性能的需求对其进行改进。

第一个，减少延迟，增加效率，通过查询网络资料得知，我们可以选择 ACK 累积确认+减少锁竞争 + 自适应重传定时器的方式来进行改进。ACK 累积确认（Cumulative ACK）是指不再每个分片都确认，而是告诉发送方“你收到的最大连续序号是多少”。发送方可以删除比该序号小的所有包。

对此，我对于原代码做了如下改进：

```
ack_packet = struct.pack(HEADER_FORMAT, expected_seq - 1)
sock.sendto(ack_packet, sender_addr)
```

使用 ACK 累积确认机制，通过累计 ACK 确认多个包，加速确认流程。接收方始终回传 expected_seq - 1，避免逐个 ACK。

```
if retries >= MAX_RETRIES:
    del send_buffer[seq]  # 丢弃该包，避免一直卡住
```

限制延迟扩大，防止单个包失败拖慢整体进度。

```
if seq > expected_seq:
    recv_buffer[seq] = payload
```

（接收方）提前缓存乱序包

```
while expected_seq in recv_buffer:
    print(f"[RECEIVER] 连续接收 seq {expected_seq}")
    del recv_buffer[expected_seq]
    expected_seq += 1
```

接收方在每次接收到期望包后，顺序处理缓存区

然后，我们想办法提高最大吞吐量：

```
WINDOW_SIZE = 20  # 原来是 5
```

最开始能想到的就是加大滑动窗口 WINDOW_SIZE 原始代码窗口为 5，限制了并发发送的数量。增加为 WINDOW_SIZE = 20 后可显著提升并发数据发送量。

```
while len(send_buffer) < WINDOW_SIZE and current_index < total_chunks:
    seq = next_seq
    next_seq += 1
    send_packet(seq, chunks[current_index])
    current_index += 1
```

其次，我们可以使用非阻塞批量发送（无需等待 ACK 再发），填满窗口再等 ACK，而不是发一个等一个，提升并发度。

然后，我们提高平均吞吐量：

```
ack_thread = threading.Thread(target=ack_listener, daemon=True)
ack_thread.start()
```

增加快速 ACK 处理机制，即 ACK Listener 线程。

```
with buffer_lock:
    keys = list(send_buffer.keys())
    for s in keys:
        if s <= ack_seq:
            del send_buffer[s]
```

在 ack_listener() 中及时移除已确认包。确保窗口及时腾空，避免阻塞后续发送。

```
time.sleep(0.05)  # 原来是 0.1，缩短检查间隔
```

超时重传优化，间隔更短、检查更频繁。

之后，我们关注到随机错误，突发错误的分析和解决。

首先，我们要知道什么是随机错误和突发错误。通过网上搜集资料得知：

随机错误指的是数据包偶尔丢失，位置不固定，误码较少但频繁。而突发错误指的是连续丢包或大范围错误，集中在短时间内发生，如网络断流、交换拥塞等。

应对“随机错误”，我们可以选择通过定期检查 send_buffer 中是否有超时未被确认的包进行重发。这一点在上面我们对效率的改进那里已经实现。其次，我们可以使用快速重传机制。

```
ack_count = defaultdict(int)


if ack_seq == last_ack:
    ack_count[ack_seq] += 1
    if ack_count[ack_seq] >= 3:
        # 触发快速重传
        resend_packet(ack_seq + 1)
else:
    last_ack = ack_seq
    ack_count[ack_seq] = 1
```

减少对单个包的等待时间，提高随机丢包处理效率。

对于突发错误，我们可以选择使用选择性确认（Selective ACK）机制。即相比简单累积 ACK（只确认连续的包），选择性确认允许接收方告诉发送方它收到了哪些“离散包”。

```
HEADER_FORMAT = "!I100B"

ack_bitmap = [1 if seq in received else 0 for seq in window_range]
ack_packet = struct.pack(HEADER_FORMAT, base_seq, *ack_bitmap)
```

注意，这里我们假设了最多 100 个窗口大小。

还可以使用动态窗口调整（拥塞控制策略）来进行改进，针对突发错误可能反映的带宽/拥塞问题，引入滑动窗口动态调整机制（如 TCP Reno 思想）：

```
if 3 次重传失败:
    WINDOW_SIZE = max(1, WINDOW_SIZE // 2)
elif 连续成功ACK:
    WINDOW_SIZE += 1  # 或指数增长
```

我们还可以引入冗余编码（前向纠错）机制。

```
# 用XOR简单校验构造冗余块
parity_block = reduce(lambda a, b: bytes([x ^ y for x, y in zip(a, b)]), data_blocks)
```

加入冗余包（如每 5 个包附加 1 个校验包），部分丢包可在接收端恢复，接收端通过校验包恢复一个丢失的数据块。

第三个任务，我们要测量 udp 协议下的最大吞吐量，吞吐量的计算公式如下：

$$\text{吞吐量 (Mbps)} = \frac{\text{总数据量 (bits)}}{\text{总时间 (seconds)}}$$

我们对发送方和接收方的代码进行修改，使发送方发送大量数据给接收方，接收方同时记录接收数据量以及耗时，然后根据上面的公式计算得出吞吐量。

```
end_time = time.time()
duration = end_time - start_time if start_time else 0.(
throughput = received_bytes / (1024 * 1024) / duration

print(f"[RECEIVER] 接收完成：")
print(f" - 接收总字节数：{received_bytes} bytes")
print(f" - 耗时：{duration:.4f} 秒")
print(f" - 吞吐量：{throughput:.4f} MB/s")
```

接收方代码修改部分。

```
HEADER_FORMAT = "!I"
HEADER_SIZE = struct.calcsize(HEADER_FORMAT)
PAYLOAD = b"hello" * 20  # 每个数据包100字节
TOTAL_PACKETS = 10000
```

发送方修改部分。

```
┌──(kali㉿kali)-[~/桌面/sy7]
└─$ python3 fs1.py
[SENDER] 开始发送 10000 个数据包到 ('175.10.107.61', 5005)
[SENDER] 发送完成：
 - 总数据量：0.95 MB
 - 耗时：0.2201 秒
 - 吞吐量：4.3322 MB/s
```

```
oslab@oslab-virtual-machine:~/桌面$ python3 js1.py
[RECEIVER] 正在监听 0.0.0.0:5005
[RECEIVER] 接收完成:
 - 接收总字节数: 1000000 bytes
 - 耗时: 0.2198 秒
 - 吞吐量: 4.3383 MB/s
```

于是我们得到结果大致为 4.33MB/s 左右。

最后一个任务：尝试为 1 中的要求设计一到多的广播协议。想要实现该协议，我们需要应对 UDP 本身不可靠 + 无连接的挑战，尤其在多个接收方参与的情况下，我们可能会面临以下问题：数据包丢失和乱序；ACK 风暴（多个接收方同时回复确认）；接收方异步状态管理。我们可以尝试着设计协议组件，大致如下：

1. 广播方式：

使用 socket.setsockopt(SOL_SOCKET, SO_BROADCAST, 1) 启用 UDP 广播；

目标地址为 192.168.X.255（广播地址）。

2. ACK 收集机制：

每个数据包发送后，发送方等待所有接收方的 ACK；

设置超时，若部分接收方未应答，则重发。

3. ACK 抑制机制（避免 ACK 风暴）：

接收方在随机延迟后发送 ACK；

可使用抑制窗口避免 ACK 同步。

4. 终止协议：

最后发送一个 FIN 包，所有接收方回 FIN_ACK；

全部 FIN_ACK 收到后终止。

但是尝试着修改原代码时发现，因为原代码实现功能本身比较复杂的原因，我们

想要实现广播式协议的话其中要增添部分过于复杂，所以我只能设计一个简易的广播式协议，即其中没有达到任务 1 中要求的所有较为复杂的功能。

```python
import socket
import struct
import time
import random

BROADCAST_IP = '192.168.219.255'  # 网段的广播地址
PORT = 5005
HEADER_FORMAT = '!I'
PAYLOAD = b'hello' * 20
TOTAL_PACKETS = 100
RECEIVERS = {'A', 'B', 'C'}  # 接收方标识

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
sock.settimeout(0.5)

def pack(seq, payload):
    return struct.pack(HEADER_FORMAT, seq) + payload

for seq in range(1, TOTAL_PACKETS + 1):
    packet = pack(seq, PAYLOAD)
    attempts = 0
    received_acks = set()
    while len(received_acks) < len(RECEIVERS) and attempts < 5:
        sock.sendto(packet, (BROADCAST_IP, PORT))
        try:
            while True:
                data, addr = sock.recvfrom(1024)
                ack_seq, receiver_id = struct.unpack('!IB', data)
                if ack_seq == seq:
                    received_acks.add(chr(receiver_id))
        except socket.timeout:
            attempts += 1
    print(f"seq {seq} acked by: {received_acks}")
sock.sendto(b'FIN', (BROADCAST_IP, PORT))
```

广播式发送方，其中我们使用了主发送循环，构建每一个数据包并发送，每个包等待所有接收方的 ACK，才进入下一个包。

```python
for seq in range(1, TOTAL_PACKETS + 1):
    packet = pack(seq, PAYLOAD)
    attempts = 0
    received_acks = set()
```

接收 ACK 以及去重和重传机制，每发送一个数据包，进入 ACK 监听环节，每个 ACK 包含一个序号 ack_seq 和一个接收方标识 receiver_id（如 A/B/C），超时未收齐就自动重传。

```
    while len(received_acks) < len(RECEIVERS) and attempts < 5:
        sock.sendto(packet, (BROADCAST_IP, PORT))
        try:
            while True:
                data, addr = sock.recvfrom(1024)
                ack_seq, receiver_id = struct.unpack('!IB', data)
                if ack_seq == seq:
                    received_acks.add(chr(receiver_id))
        except socket.timeout:
            attempts += 1
    print(f"seq {seq} acked by: {received_acks}")
```

最后广播一次 FIN 表示传输结束，所有接收方检测到后退出。

```
sock.sendto(b'FIN', (BROADCAST_IP, PORT))
```

接收方代码:

```
import socket
import struct
import random
import time

MY_ID = b'A'
PORT = 5005
HEADER_FORMAT = '!I'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', PORT))
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

while True:
    data, addr = sock.recvfrom(1024)
    if data == b'FIN':
        break
    if len(data) >= 4:
        seq = struct.unpack(HEADER_FORMAT, data[:4])[0]
        # 模拟延迟ACK
        time.sleep(random.uniform(0.01, 0.1))
        ack = struct.pack('!IB', seq, ord(MY_ID))
        sock.sendto(ack, addr)
```

绑定所有本地地址（监听广播）；启用广播支持。

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', PORT))
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
```

接收方解析数据包的序号，模拟 随机 ACK 延迟（用于避免 ACK 风暴），发送 ACK 回广播源（注意，UDP 允许对广播源响应）。

```
data, addr = sock.recvfrom(1024)
if data == b'FIN':
    break
if len(data) >= 4:
    seq = struct.unpack(HEADER_FORMAT, data[:4])[0]
    # 模拟延迟ACK
    time.sleep(random.uniform(0.01, 0.1))
    ack = struct.pack('!IB', seq, ord(MY_ID))
    sock.sendto(ack, addr)
```

四.结论

本次实验本次实验围绕以下核心目标展开：实现一对一可靠传输协议，使发送方与接收方最终都能明确每个数据包的状态（成功接收或最终失败）。提升协议性能，包括减少延迟、提升吞吐量、增强对网络误差的适应能力。基于 UDP 协议进行实现，并测量协议在真实网络环境（如宿舍）中的最大吞吐量。拓展到一对多广播通信协议的设计与初步实现。

协议设计的本质是权衡可靠性与效率。UDP 的无连接与不可靠特性，虽然提供了灵活性，但也对应用层协议提出了更高要求。

滑动窗口与 ACK 策略的优化对于性能提升至关重要，尤其在丢包环境下，使用 SACK 比简单 ACK 明显更稳定。状态同步机制与最终确认逻辑是本实验的难点。如何让双方“都知道彼此知道最终状态”是设计中的核心问题。广播协议设计的挑战远高于点对点传输，尤其在于接收方的异步状态管理与 ACK 聚合策略。实际网络测试的重要性不可忽视，理论设计往往理想化，实际网络中误码、丢包、延迟、ACK 聚合问题非常真实。

参考文献

1. https://blog.csdn.net/u011563903/article/details/90116368. 详解一次完整的数据包传输过程 -- 层层递进. shufanhao.top 2023.11.25.

[2] https://blog.csdn.net/a595364628/article/details/64123565. 网络中的三种通讯模式：单播、广播、组播(多播). a595364628.2024.11.14.

[3] https://cloud.tencent.com/developer/article/2362927 网络协议性能优化：从 HTTP 到 TCP、DNS、SSL/TLS 的全面探究. bug 菌.2021.01.24.

[4] https://developer.aliyun.com/article/1393349. 网络协议之性能优化与性能评估.Public IP List.2023.8.15.

[5] https://www.tance.cc/article/know/432. 网络协议栈优化：提升高并发环境下的数据传输效率. tance.cc.2022.3.24

[6] https://blog.csdn.net/weixin_42839065/article/details/131375292. 干吃咖啡豆. 【TCP/IP】广播 - 定义、原理及编程实现. 2024.9.16.

[7] https://blog.csdn.net/xiaolei251990/article/details/83177165. 用 udp 实现广播通信. 【鹰击司马】. 2022.8.15.