南开大学
Nankai University

南　　开　　大　　学

计算机网络实验报告

实验 3-2

姓名：李雅帆

学号：2213041

年级：2022 级

专业：信息安全

2024 年 12 月 7 日

目录

一、 实验要求

在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，发送窗大于 1，接收窗口大小为 1，支持累积确认，完成给定测试文件的传输。

- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 流水线协议：多个序列号
- 发送缓冲区、接收缓冲区
- 累积确认：Go Back N
- 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

二、 协议设计

(一) 滑动窗口

滑动窗口是一种流量控制机制，用于在数据传输中动态管理发送端和接收端之间的数据流量。通过限制发送端未被确认的数据包数量，滑动窗口机制可以在提高传输效率的同时确保接收端缓冲区不溢出。

1. 滑动窗口的关键点

   - 窗口范围:
     - base：已被确认的最后一个数据包的序号。
     - nextseqnum：下一个将要发送的数据包序号。
     - 发送窗口范围：$[\text{base}+1, \text{nextseqnum}-1]$
   - 窗口的移动: 当接收端发送 ACK 时，更新 base 值，窗口向前滑动，释放空间允许发送更多数据包。
   - 窗口大小: 窗口大小由网络条件和接收端的缓冲区容量决定。本实验中，窗口大小设置为固定值 Windows = 10。

2. 滑动窗口的实现

   - 发送端逻辑：发送端根据窗口范围发送数据包，接收到 ACK 后更新窗口。

```
if ((nextseqnum - 1) - (base + 1) < Windows && nextseqnum != package_num) {
    int len = (nextseqnum == package_num - 1) ? (datasize - (package_num - 1)
        * MAXSIZE) : MAXSIZE;
    int seqnum = nextseqnum % 256;

    if (send_package(client, server_addr, serveraddr_len, data_content +
        nextseqnum * MAXSIZE, len, seqnum) == -1) {
        nextseqnum--;  // 发送失败，重发当前包
    } else {
        if (nextseqnum == base + 1) {
            start = clock();  // 启动定时器
        }
        nextseqnum++;  // 成功发送，更新窗口右边界
    }
}
```

   - 接收端逻辑：接收端判断收到的数据包是否按序，并发送累积确认。

```
if (seq_predict == int(recvpkt->get_ack())) {
    memcpy(data + file_len, recvpkt->get_data_content(), recvpkt->
        get_datasize());
    file_len += recvpkt->get_datasize();
    seq = seq_predict;
    seq_predict = (seq_predict + 1) % 256;

    header.set_tag(ACK);
```

```
        header.set_ack((u_char)seq);
        header.set_sum(cksum((u_short*)&header, sizeof(header)));
        sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr,
            clientaddr_len);
    } else {
        // 非按序数据包，丢弃并发送上一个确认号
    }
```

- 窗口移动：滑动窗口根据接收到的 ACK 进行更新。

```
if (header.get_tag() == ACK && cksum((u_short*)&header, sizeof(header)) == 0)
    {
    base += (int(header.get_ack()) - (base + 1) % 256 + 1);
    start = clock();  // 重置定时器
}
```

3. 实验现象

- 窗口滑动：接收到 ACK 后，base 值更新，滑动窗口向前移动。
- 数据重传：窗口中未确认的数据包超时后，重新发送。
- 窗口限制：发送窗口大小为 10，超过后必须等待 ACK 才能继续发送新数据。

4. 滑动窗口的优点

- 提高传输效率：允许发送端在等待 ACK 的同时继续发送数据，提高网络带宽利用率。
- 流量控制：避免发送端发送过多数据导致接收端缓冲区溢出。
- 丢包容错：窗口内未确认的数据包可以通过重传机制修复。
- 减少延迟：减少了每发送一个数据包都等待 ACK 的传输延迟。

(二) 退回 N 帧协议

退回 N 帧（Go-Back-N, GBN）协议是一种可靠的数据传输协议，允许发送端连续发送多个数据包，但在接收端检测到错误或丢失时，会丢弃当前及后续的所有未确认数据包，并要求发送端从出错的数据包重新发送。

1. 退回 N 帧协议的实现

- 窗口机制：
  - 发送窗口：发送端维护一个窗口，控制允许发送但未确认的数据包数量。
  - 接收窗口：接收端只接受按序到达的数据包，乱序包被丢弃。
  - 在本代码中，发送窗口大小设置为固定值 Windows = 10
- 数据包的发送与确认：
  - 发送端连续发送窗口内的数据包，不需要逐包等待 ACK。
  - 接收到 ACK 后，窗口向前滑动，释放被确认的数据包位置。
- 错误处理：如果接收端未按序接收到数据包或校验和验证失败，则丢弃该包并重传窗口内所有未确认的数据包。

2. 代码中的实现

- 发送端逻辑：发送端维护 base 和 nextseqnum，其中 base 表示已确认的最后一个数据包,nextseqnum 表示将要发送的下一个数据包序号。

```
while (base != package_num - 1) {
    if ((nextseqnum - 1) - (base + 1) < Windows && nextseqnum != package_num)
        {
        int len = (nextseqnum == package_num - 1) ?
                  (datasize - (package_num - 1) * MAXSIZE) : MAXSIZE;
        int seqnum = nextseqnum % 256;

        if (send_package(client, server_addr, serveraddr_len, data_content +
            nextseqnum * MAXSIZE, len, seqnum) == -1) {
            cout << "[发送数据包失败]" << endl;
            nextseqnum--;  // 发送失败，重发当前包
        } else {
            if (nextseqnum == base + 1) start = clock();  // 初始化定时器
            nextseqnum++;  // 成功发送，窗口右端向前移动
        }
    }

    // 检查 ACK
    if (recvfrom(client, recv_buffer, sizeof(header), 0, (SOCKADDR*)&
        server_addr, &serveraddr_len) != -1) {
        memcpy(&header, recv_buffer, sizeof(header));
        if (header.get_tag() == ACK && cksum((u_short*)&header, sizeof(header
            )) == 0) {
            base += (int(header.get_ack()) - (base + 1) % 256 + 1);  // 更新
                base
            start = clock();  // 重置定时器
        }
    } else if (clock() - start > MAX_TIME) {
        cout << "[超时]重传窗口内数据包" << endl;
        nextseqnum = base + 1;  // 超时，退回未确认数据包
    }
}
```

- 接收端逻辑: 接收端只接受按序到达的数据包，乱序包被丢弃并返回上一个已确认的数据包的 ACK。

```
if (seq_predict != int(recvpkt->get_ack())) {
    cout << "[乱序数据包]序号:" << recvpkt->get_ack() << ", 丢弃数据包" <<
        endl;
    header.set_tag(ACK);
    header.set_ack((u_char)seq);  // 返回上一个已确认的序号
    header.set_sum(cksum((u_short*)&header, sizeof(header)));
    sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr,
        clientaddr_len);
    continue;
```

```
}

// 按序接收的包，存储数据并返回确认
memcpy(data + file_len, recvpkt->get_data_content(), recvpkt->get_datasize());
file_len += recvpkt->get_datasize();
seq = seq_predict;
seq_predict = (seq_predict + 1) % 256;
header.set_tag(ACK);
header.set_ack((u_char)seq);
header.set_sum(cksum((u_short*)&header, sizeof(header)));
sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr,
    clientaddr_len);
```

3. 实验现象

- 正常传输: 数据包按序到达，接收端发送 ACK，窗口平稳滑动。
- 乱序数据包: 接收端检测到乱序，丢弃数据包并返回上一个已确认的序号。
- 丢包与重传: 接收端未收到数据包，发送端检测超时，重传窗口内未确认的数据包。

4. 退回 N 帧的特点

- 错误检测与重传: 接收端只能按序接收数据包，检测到乱序或损坏数据包后，丢弃当前及后续所有数据包; 发送端重传窗口内所有未确认的数据包。
- 高效性: 连续发送窗口内的数据包，无需等待逐个 ACK，提高了传输效率。
- 简单性: 接收端逻辑简单，只需判断当前包是否按序到达。
- 适合高丢包环境: 在高丢包率下，GBN 协议可快速发现错误并重传丢失数据。

(三) 累积确认

累积确认是可靠数据传输协议中的一种确认机制，接收端通过发送确认号（ACK），告知发送端已成功接收的最大连续数据包序号。累积确认避免了逐一确认每个数据包，从而减少了通信开销，提高了传输效率。

1. 累积确认的关键点

- 确认范围: 接收方发送的 ACK 表示其已正确接收到的所有连续数据包的最后一个序号。接收端通过校验接收到的数据包序号，判断是否按序。如果按序，更新序列号并发送累积确认；如果乱序，仅确认前面连续接收的数据包。
- 避免乱序重传: 接收方如果收到不连续的数据包，只会确认之前连续的数据。
- 减少冗余 ACK: 只需发送一次 ACK 确认多个连续数据包，而无需对每个数据包单独发送 ACK。

2. 实验现象

- 按序接收: 数据包按序到达时，接收端逐一累积确认；发送端接收确认后，滑动窗口平稳移动。

- 乱序接收：数据包乱序到达时，接收端只确认连续数据包，并丢弃乱序包；发送端通过超时或接收端请求重传丢失数据包。
- 丢包重传：丢包时，接收端累积确认丢包前的所有数据；发送端超时重传未被确认的数据包。

3. 累积确认在代码中的实现

- 接收 ACK: 在接收到数据包的 ACK 后，检查是否为累积确认；更新发送窗口的起点 base 到 ACK 所确认的最大连续数据包。
- 重传未确认数据包: 如果定时器超时且未收到对应的 ACK，重传发送窗口内所有未确认的数据包。

```
// 发送端累积确认处理
if (recvfrom(client, recv_buffer, sizeof(header), 0, (SOCKADDR*)&server_addr,
    &serveraddr_len) != -1) {
    memcpy(&header, recv_buffer, sizeof(header));
    if (header.get_tag() == ACK && cksum((u_short*)&header, sizeof(header))
        == 0) {
        cout << "收到确认ACK:" << endl;
        header.print_header();

        // 更新 base 值
        if (int(header.get_ack()) == (base + 1) % 256) {
            base += (int(header.get_ack()) - (base + 1) % 256 + 1);
            start = clock(); // 重置定时器
        }
    }
} else {
    // 超时逻辑
    if (clock() - start > MAX_TIME) {
        cout << "[超时]未收到ACK, 重传窗口内数据包" << endl;
        nextseqnum = base + 1;
    }
}
```

```
// 接收端累积确认处理
if (seq_predict == int(recvpkt->get_ack())) {
    // 按序接收, 存储数据并更新序列号
    memcpy(data + file_len, recvpkt->get_data_content(), recvpkt->
        get_datasize());
    file_len += recvpkt->get_datasize();
    seq = seq_predict;
    seq_predict = (seq_predict + 1) % 256;

    // 返回累积确认
    header.set_tag(ACK);
    header.set_ack((u_char)seq);
    header.set_sum(cksum((u_short*)&header, sizeof(header)));
```

```
    sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr,
        clientaddr_len);
}
```

4. 累积确认的优点

- 减少 ACK 数量: 一次确认多个数据包，降低通信开销。
- 提高效率: 减少频繁 ACK 发送对带宽的占用。
- 简化逻辑: 接收端仅需记录最大连续数据包序号。
- 容忍丢包: 丢失部分数据包时仍可确认前序数据包。
- 兼容滑动窗口: 提升传输效率，动态调整发送窗口。

（四） 超时重传机制

超时重传是确保可靠数据传输的重要手段。当发送方在一定时间内未收到接收方的确认（ACK）时，假定对应的数据包可能丢失，触发重传机制。重传机制结合定时器用于判断数据是否超时，从而实现对数据丢失的自动修复。

1. 超时重传的关键点:

- 定时器机制: 每次发送窗口的第一个数据包时，启动定时器，定时器的超时时间设置为 MAX_TIME，在代码中等于 $0.5 \times$ CLOCKS_PER_SEC，即 0.5 秒。
- 重传条件: 如果定时器超时且未收到 ACK，重传窗口内所有未确认的数据包；重置定时器，继续等待确认。
- 重传范围: 仅重传窗口内未确认的数据包，而不是所有已发送数据包；发送窗口通过 base 和 nextseqnum 来控制，base + 1 到 nextseqnum - 1 是未确认的数据包范围。

2. 超时重传在代码中的实现:

- 定时器的实现：在每次发送窗口的第一个数据包时，初始化定时器 start。使用 clock() 函数记录当前时间，与 start 比较，若超出MAX_TIME则认为超时。

```
clock_t start = clock();  // 启动定时器
if (clock() - start > MAX_TIME) {  // 检查是否超时
    // 触发超时处理逻辑
}
```

- 重传逻辑：如果定时器超时，重传窗口中所有未确认的数据包。

```
if (clock() - start > MAX_TIME) {
    cout << "[超时]␣重传窗口内数据包:" << endl;

    for (int i = base + 1; i < nextseqnum; i++) {
        int seqnum = i % 256;  // 确保序列号在 [0, 255] 内循环
        int len = (i == package_num - 1) ? (datasize - (package_num - 1) *
            MAXSIZE) : MAXSIZE;

```

```
        if (send_package(client, server_addr, serveraddr_len, data_content +
            i * MAXSIZE, len, seqnum) == -1) {
            cout << "[重传失败]第" << seqnum << "号数据包" << endl;
        } else {
            cout << "[重传成功]第" << seqnum << "号数据包" << endl;
        }
    }

    start = clock();  // 重置定时器
}
```

- 非阻塞模式的辅助
  - 避免在 recvfrom() 函数中因阻塞造成的程序卡顿，允许在超时检测期间执行其他逻辑。
  - 使用 ioctlsocket() 将套接字设置为非阻塞模式。
  - 在非阻塞模式下，recvfrom() 返回 -1 时表示当前没有接收到数据，而不是等待数据到达。

```
u_long mode = 1;
ioctlsocket(socket, FIONBIO, &mode);
```

- 在完成超时检测后，可以将套接字恢复为阻塞模式以确保后续接收的稳定性。示例：

```
mode = 0;
ioctlsocket(socket, FIONBIO, &mode);
```

3. 实验现象
   - 正常传输: 数据包按序到达并被确认，定时器重置。
   - 丢包现象: 数据包丢失时，发送端检测到超时，重传未被确认的数据包。
   - 网络抖动: 数据包延迟导致超时触发，重传后接收到原数据包和重传包。
4. 超时重传的优点
   - 保证数据完整性: 通过重传机制确保丢失的数据包能够重新发送。
   - 减少传输延迟: 使用定时器结合非阻塞模式，可快速检测丢包并执行重传。
   - 适配网络波动: 超时机制可以应对网络抖动、丢包等情况，提升传输的鲁棒性。

（五） 数据包 Header 结构

字段大小总计 40 位（5 字节），用于控制和校验数据包，各字段用途如下。

字段名称	位数	描述
数据长度	16 位	表示数据部分的长度，以字节为单位，用于接收端解析数据。
校验和	16 位	用于检测数据包在传输过程中的完整性。
FIN	1 位	标志断开连接请求，若置为 1，表示请求断开连接。
ACK	1 位	确认标志，若置为 1，表示该包为确认包。
SYN	1 位	同步标志，若置为 1，表示请求建立连接。
序列号/确认号	8 位	若为序列号，用于标识当前数据包；若为确认号，表示对上一数据包的确认。

图 1: Header

```
class Header {
        /*
        根据数据长度选择类型:
        char-8位;short-16位;int-32位;long-32/64位;long long-64位
        */
        unsigned short datasize = 0;
        unsigned short sum = 0;
        unsigned char tag = 0;
        unsigned char ack = 0;
 public:
        Header() :datasize(0), sum(0), tag(0), ack(0) {};
        void set_tag(unsigned char tag) {
                this->tag = tag;
        }
        void clear_sum() {
                sum = 0;
        }
        void set_sum(unsigned short sum) {
                this->sum = sum;
        }
        unsigned char get_tag() {
                return tag;
        }
        unsigned short get_sum() {
                return sum;
        }
        void set_datasize(unsigned short datasize) {
                this->datasize = datasize;
        }
        void set_ack(unsigned char ack) {
                this->ack = ack;
        }
```

```
        int get_datasize() {
                return datasize;
        }
        unsigned char get_ack() {
                return ack;
        }
        void print_header() {
                printf("datasize:%d, ack:%d, tag:%d\n", get_datasize(),
                    get_ack(), get_tag());
        }
};
```

(六) 数据包 Packet 结构

- 数据包总大小计算
- 如果有要传输的数据的时候就放在 char 数组里面，同时对头部的数据长度进行赋值。
- 如果没有要传输的数据的时候（比如发送确认包 ACK 或者 SYN 包，FIN 包等，不带数据时可以只发送头部）

1. 数据包总大小计算, 数据包的总大小由头部大小和数据部分大小决定。
2. 数据内容操作: 设置数据内容并获取数据内容。
3. 数据长度控制: 通过头部字段 datasize 指定数据部分的长度。
4. 校验和设置与校验: 在发送数据包时，计算并设置校验和, 在接收数据包时，使用校验和验证完整性。
5. 打印数据包信息, 用于调试时输出数据包的详细内容。

```
class Packet {
private:
        Header header;
        char data_content[1024];
public:
        Packet() :header() { memset(data_content, 0, 1024); }
        Header get_header() {
                return header;
        }
        void set_datacontent(char* data_content) {
                memcpy(this->data_content, data_content, sizeof(data_content)
                    );
        }
        int get_size() {
                return sizeof(header) + header.get_datasize();
        }
        void print_pkt() {
```

```
17                  printf("bytes:%d\nseq:%d\n\n", header.get_datasize(), header.
                        get_ack());
18          }
19          char* get_data_content() {
20                  return data_content;
21          }
22          void set_datasize(int datasize) {
23                  header.set_datasize(datasize);
24          }
25          int get_datasize() {
26                  return header.get_datasize();
27          }
28          void set_ack(unsigned char ack) {
29                  header.set_ack(ack);
30          }
31          u_char get_ack() {
32                  return header.get_ack();
33          }
34          void set_tag(unsigned char tag) {
35                  header.set_tag(tag);
36          }
37          void clear_sum() {
38                  header.clear_sum();
39          }
40          void set_sum(unsigned short sum) {
41                  header.set_sum(sum);
42          }
43          unsigned char get_tag() {
44                  return header.get_tag();
45          }
46          unsigned short get_sum() {
47                  return header.get_sum();
48          }
49          void printpacketmessage() {
50                  printf("Packet␣size=%d␣bytes, tag=%d, seq=%d, sum=%d,
                        datasize=%d\n", get_size(), get_tag(), get_ack(), get_sum
                        (), get_datasize());
51          }
52  };
```

（七） 差错检验

本实验的差错检验采用经典的 16 位校验和算法（Checksum）。在数据传输过程中，使用校验和对数据包进行完整性检查，接收端通过验证校验和判断数据是否被篡改或损坏。

1. 校验和计算的原理。

   - 将数据划分为若干个 16 位块（2 字节）。

- 将所有 16 位块逐块相加，产生一个 16 位的累加和。
- 如果累加和溢出（超过 16 位），则将高位的溢出部分加回到低位。
- 对累加和取反，得到校验和。
- 在数据传输时，将计算得到的校验和附加到报文中。
- 接收方重新计算收到数据的校验和。
- 将接收到的校验和与计算结果相加，验证总和是否为 0xFFFF。
- 若结果为 0xFFFF，说明数据完整；否则认为数据损坏。

cksum 函数来实现 16 位校验和的计算：

```
u_short cksum(u_short* message, int size) {
    int count = (size + 1) / 2;
    u_long sum = 0;

    unsigned short* buf = (unsigned short*)malloc(size);
    memset(buf, 0, size);
    memcpy(buf, message, size);

    // 遍历每个 16 位块进行累加
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    // 对累加和取反，得到最终的校验和
    u_short result = ~(sum & 0xffff);
    return result;
}
```

发送端设置校验和: 在发送数据包前，计算校验和并存储在头部。

```
header.clear_sum();  // 先清零校验和字段
header.set_sum(cksum((u_short*)&header, sizeof(header)));  // 计算并设置校验和
```

接收端验证校验和: 接收端收到数据包后，重新计算校验和进行校验。

```
if (cksum((u_short*)&header, sizeof(header)) == 0) {
    cout << "[校验成功]数据包完整" << endl;
} else {
    cout << "[校验失败]数据包损坏，丢弃" << endl;
    continue;  // 丢弃损坏的数据包
}
```

2. 差错检验的使用场景

- 三次握手和四次挥手中的校验: 在三次握手和四次挥手过程中，每个报文都附带校验和，用于检测控制报文的完整性。
- 数据传输中的校验: 在文件数据包的传输过程中，每个数据包都计算校验和并附加到报文头部, 接收方对收到的数据包进行校验，若校验失败则丢弃该包并等待重传。

3. 差错处理

- 控制报文的校验失败：若握手或挥手报文校验失败，接收方丢弃该报文并等待重传。例如，服务器未正确接收到客户端的 SYN 报文时：

```
if (header.get_tag() != SYN || cksum((u_short*)&header, sizeof(header)) != 0)
    {
    cout << "[接收失败]␣报文校验错误，丢弃" << endl;
    continue;
}
```

- 数据包的校验失败：若接收到的数据包校验和错误，接收方直接丢弃该包，发送方在超时后重新发送该数据包，直到接收方确认。例如，校验失败时的输出：

```
[校验失败] 数据包损坏，丢弃
```

（八） 三次握手建立连接

1. 客户端发送 SYN：客户端向服务器发起连接请求。
2. 服务器返回 SYN+ACK：服务器接收到请求后进行响应，表示可以建立连接。
3. 客户端发送 ACK：客户端接收服务器响应后发送确认报文，完成连接的建立。

（九） 四次挥手断开连接

1. 第一次挥手：客户端发送 FIN 请求断开连接。
2. 第二次挥手：服务器接收 FIN，并返回 ACK 确认。
3. 第三次挥手：服务器发送 FIN+ACK 请求断开连接。
4. 第四次挥手：客户端接收 FIN+ACK，并返回 ACK 确认。

三、 实现流程

(一) 初始化与套接字建立

发送端与接收端分别执行以下操作:

1. 调用 `WSAStartup()` 初始化 WinSock 库。
2. 创建 UDP 套接字 `socket(AF_INET, SOCK_DGRAM, 0)`。
3. 对接收端: 调用 `bind()` 将套接字绑定到指定的本地端口 (如 8888)。
4. 对发送端: 使用 `InetPton()` 设置服务器 IP 地址和端口 (与接收端对应)。

(二) 三次握手建立连接 (模拟 TCP)

发送端 (client):

1. 设置 Header 中 `tag=SYN`, 校验和置零后计算校验和, 将报文发送给接收端。
2. 开启计时器, 等待接收端响应。如果超时未收到, 则重传 `SYN`。

接收端 (server):

1. 阻塞等待接收数据, 收到后计算校验和, 并检查 `tag` 是否为 `SYN`。
2. 若正确, 打印“[接收]SYN”并发送 `SYN+ACK` 包回给客户端 (`tag=ACK_SYN`), 重新计算校验和后发送。

发送端 (client):

1. 切换为非阻塞模式, 等待 `SYN+ACK`。若超时重传 `SYN`。
2. 收到 `SYN+ACK` 后, 检查校验和与标志位, 如正确, 则发送 `ACK` 包表示第三次握手完成。
3. 若发送成功, 即连接建立成功。

接收端 (server):

1. 接收到第三次握手的 `ACK` 后, 连接成功。打印“连接成功”。

此时双方建立了一个“逻辑连接”, 尽管底层仍是 UDP, 但代码模拟实现了 TCP 的三次握手过程。

(三) 数据发送与接收 (可靠数据传输模拟)

发送端 (client) 发送过程:

1. **读取文件数据**: 发送端从指定文件中读取数据存入缓存 (`buffer`)。
2. **发送文件名**: 调用 `send()` 函数, 将文件名作为数据通过分片/打包过程发送给接收端。
   - 分片规则: 如果长度超过 1024, 则分包发送; 否则只发一个包。
   - 每个包的 `ack` 字段作为序列号使用 (0-255 循环), `tag` 为普通数据包时为 0 或包含 `ACK` 位 (依据代码)。

- 使用超时重传和累积确认机制：
  - base 初始为 -1，nextseqnum 从 0 开始。
  - 当 nextseqnum 在窗口允许范围内，发送 Packet 并启动计时器（由发送窗口首个未确认包启动）。
  - 如果在 MAX_TIME 内没有收到 ACK，则重传该窗口内的所有未确认包。
  - 收到 ACK 后根据返回的确认序号移动 base。

3. **发送文件数据**：同发送文件名过程类似，将实际文件内容分片发送。发送完所有数据包后，再发送 OVER 包表示数据传输结束。

4. **等待接收端确认接收到 OVER**：发送端在接收到对方返回的 OVER 包后，说明对方已接收完全部数据。

接收端（server）接收过程：

1. 在接收文件名和文件数据时，都在 recvdata() 函数中实现：
   - 循环调用 recvfrom() 接收数据包。
   - 使用 cksum() 对包进行校验和检验。
   - 丢包模拟：根据指定丢包率随机丢弃部分数据包；如果丢弃则不发送 ACK，使发送端超时重传。
   - 延时模拟：接收到包后等待指定毫秒数再处理，模拟网络延迟。
2. 序列号确认机制：
   - 维护 seq（上一个已确认包的序列号）和 seq_predict（期待下一个包的序列号）。
   - 如果收到的数据包序列号与 seq_predict 不符，则丢弃，并对上一次已确认的序列号发送 ACK，促进重传。
   - 如果序列号匹配，则保存数据，更新 seq 和 seq_predict（$(\text{seq\_predict}+1)\%256$），并发送 ACK 给发送端。
3. 当接收端收到 OVER 包且校验通过，表明数据传输结束。此时发送 OVER 回给发送端，表示接收端数据接收完成。

（四） 四次挥手断开连接（模拟 TCP）

发送端（client）：

1. 发送 FIN 包给接收端并启动计时器，等待 ACK。
2. 若超时未收到则重传 FIN。
3. 收到 ACK 后，进入等待对方的 FIN+ACK。

接收端（server）：

1. 收到 FIN 后，返回 ACK。
2. 立刻发送 FIN+ACK 给发送端，再启动超时重传机制。
3. 等待发送端的最后 ACK。

发送端（client）：

1. 收到 FIN+ACK 后发送最后的 ACK。
2. 完成四次挥手过程，断开连接。

接收端（server）：

1. 接收到最终的 ACK，连接断开。
2. 可以关闭套接字并结束程序。

（五） 实验结束

1. 发送端打印文件传输时间与吞吐率信息。
2. 接收端将数据写入本地文件，完成整个文件传输。
3. 双方调用 closesocket() 和 WSACleanup() 释放资源。

四、 程序界面展示

(一) 丢包率和时延

- 如图2所示，Router 传输不太稳定，我自己设置了丢包和延时。
- 丢包率设为 3%。
- 时延设为 5ms。

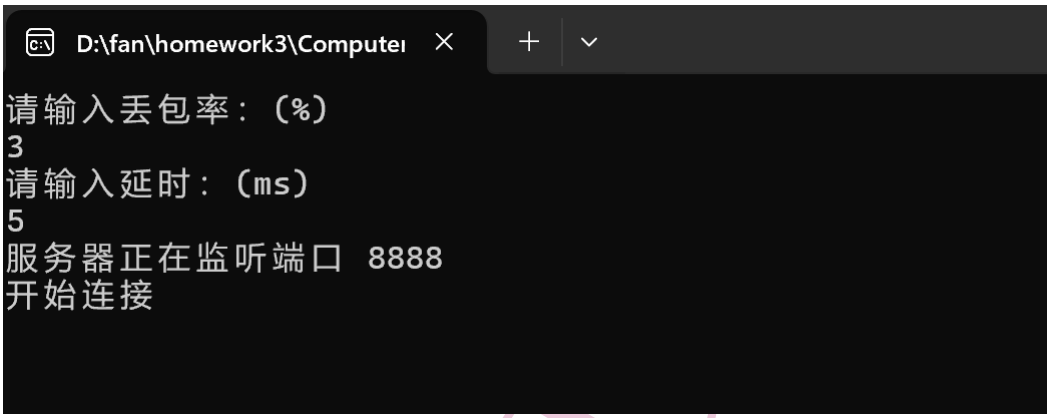

图 2: Router

(二) 启动服务器端和客户端

- 如图3所示，服务器端成功启动后会出现“服务器正在监听端口 8888”的字样。
- 客户端启动后进行三次握手连接。

图 3: 启动

(三) 发送文件

- 如图4所示，发送 1.jpg 图片文件。

图 4: 发送文件

- 客户端发送成功。

图 5: 发送成功

- 服务器端接收成功。

图 6: 接收成功

- 成功下载到本地。

图 7: 成功下载到本地

（四） 累积确认

```
即将发送当前文件中的第1813号数据包：
检查数据包内容：
Packet size=847 bytes, tag=0, seq=21, sum=41299, datasize=841
成功发送数据包：
Packet size=847 bytes, tag=0, seq=21, sum=41299, datasize=841
当前文件中的第1813号数据包发送成功！
packages: 1814
发送的数据包已经被确认：
datasize:0, ack:16
base: 1807, nextseqnum: 1814

发送的数据包已经被确认：
datasize:0, ack:17
base: 1808, nextseqnum: 1814

发送的数据包已经被确认：
datasize:0, ack:18
base: 1809, nextseqnum: 1814

发送的数据包已经被确认：
datasize:0, ack:19
base: 1810, nextseqnum: 1814

发送的数据包已经被确认：
datasize:0, ack:20
base: 1811, nextseqnum: 1814

发送的数据包已经被确认：
datasize:0, ack:21
base: 1812, nextseqnum: 1814
```

图 8: 累积确认

（五） 四次挥手断开连接

- 客户端:

图 9: 客户端四次挥手

- 服务器端:

图 10: 服务器端四次挥手

(六) 传输时间和吞吐率

图 11

五、 传输结果分析

本实验在 UDP 底层基础上模拟了 TCP 式的可靠传输过程。测试结果表明:

1. **可靠性与有序性**: 通过校验和、序列号管理和超时重传机制，接收端能够检测损坏和丢失的数据包，发送端及时重传未确认的包，实现数据完整、有序到达。
2. **累积确认与流控机制**: 接收端使用累积确认，对收到的正确有序数据包进行确认，若出现乱序或丢包则重发 ACK，促使发送端进行相应的数据重传，最终确保数据按序接收。
3. **丢包率与延迟影响**: 在设置一定丢包率和延时参数后，尽管传输耗时和吞吐率有所下降，但最终仍能成功传输完整数据，验证了该模拟过程的健壮性和可靠性。
4. **连接建立与断开**: 三次握手与四次挥手的模拟均能顺利完成，双方在逻辑层面成功建立并可靠释放连接。