计算机网络实验3-2实验报告

姓名：阿斯雅 学号：2210737

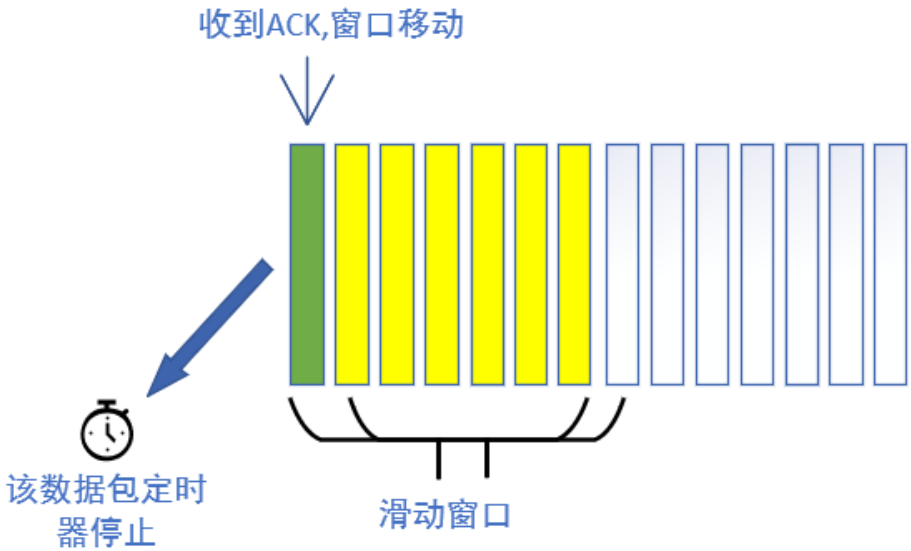
一、协议设计

实验3-1回顾

在实验3-1中，我们已经实现了基于停等机制的UDP上可靠传输的基本功能，包括连接的建立与断开、确认的接收与发送、数据的检验和计算、以及超时后的重传等操作。停等机制的核心是发送方发送一个数据包后等待接收方的确认，直到收到确认才能继续发送下一个数据包。而在实验3-2中，我们需要将这种停等机制改为滑动窗口机制，这意味着客户端和服务端之间的数据传输不再是逐包确认的，而是能够同时发送多个数据包并通过滑动窗口来控制数据的流动。为了实现这一目标，我们需要对协议的设计进行调整，特别是在客户端和服务端处理文件传输的方式上进行修改。具体来说，客户端需要在发送数据时根据窗口大小决定可发送的数据包数，而服务端需要根据接收到的数据包序列号进行累积确认。除此之外，其他功能不需要做大的修改，只需要确保在滑动窗口机制下正常工作即可。

滑动窗口机制

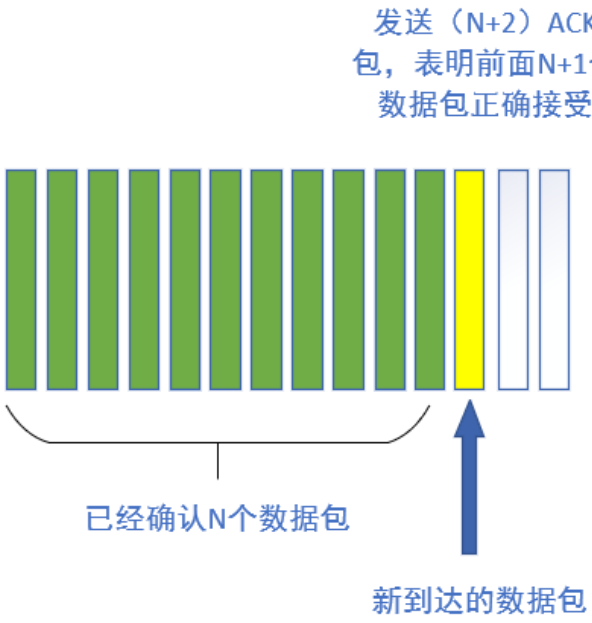
在本次实验中，客户端需要实现滑动窗口机制，具体采用的是GBN（Go-Back-N，回退N）协议。根据GBN协议，客户端维护一个滑动窗口，窗口中可以包含三种类型的数据包：已确认的数据包、未确认的数据包和未发送的数据包。初始时，客户端会将窗口内所有的数据包一次性发送给服务端，并进入等待状态，等待服务端的确认。每当客户端收到一个期望的ACK确认包时，它就会根据该确认包的序列号移动窗口，向前滑动一个位置，同时继续发送新的数据包。在此过程中，客户端会不断根据滑动窗口的大小和收到的确认包来控制数据的发送和确认，确保数据的可靠传输。



要注意的是，若出现数据包丢失的情况，客户端会通过超时重传机制来确保数据包能够最终被成功传输到服务端。但因为已经发出去了好几个数据包，所以在本次实验中我给每个数据包都设置了一个定时器，直到收到该数据包的累积确认定时器才停止，否则对应数据包的定时器就会超时重传。

累积确认机制

由于客户端采用了滑动窗口机制，服务端也必须采用相应的机制来适配这一变化。为此，服务端需要实现累积确认机制。在实验3-1中，服务端的确认机制是针对每个收到的数据包进行单独确认，即每接收到一个数据包就立即发送一个确认包。而在本次实验中，服务端需要进行累积确认，这意味着当服务端接收到一个数据包时，它不仅会确认当前接收到的数据包，还会确认所有之前已经成功接收的数据包。因此，服务端会发送一个累积确认的ACK包，表示它已经正确接收了客户端发送的当前数据包及之前所有数据包。这样，客户端就能知道哪些数据包已经被成功接收，并根据这个信息调整滑动窗口的位置。



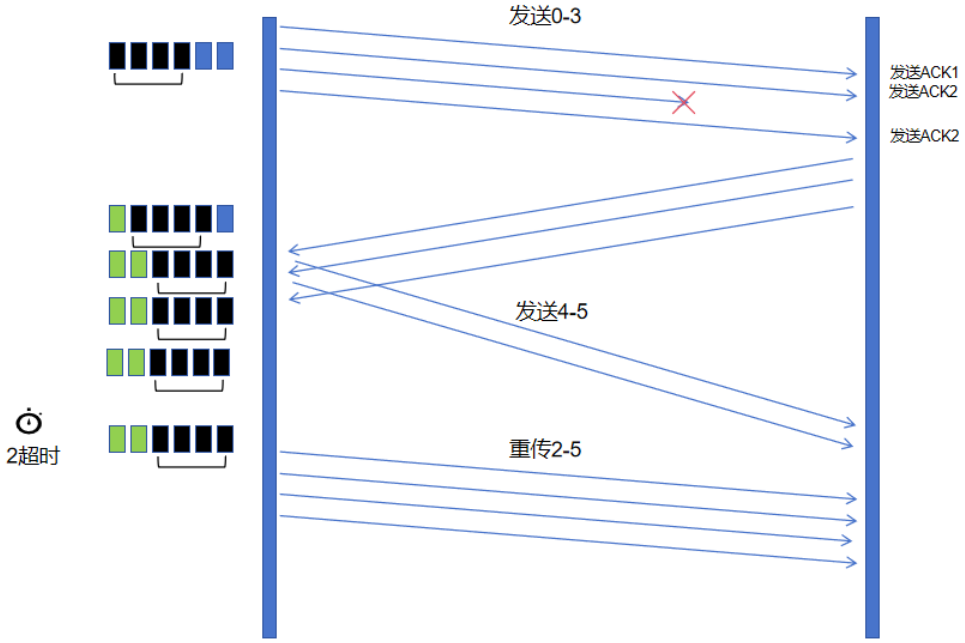
异常情况处理

接下来就是可能出现的异常情况及相应的解决办法了。

客户端丢包

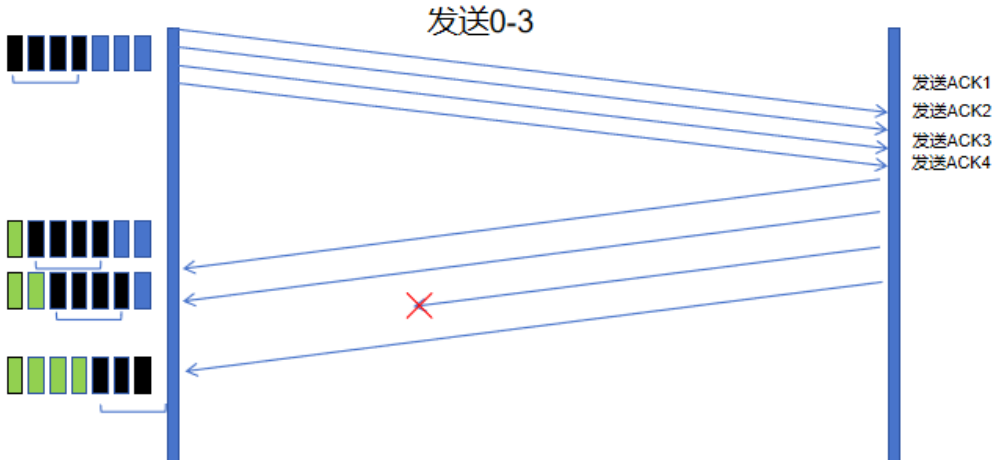
第一种常见的情况是客户端丢包。当客户端丢包时，服务端可能会收到乱序的数据包，因为某些数据包并未成功到达客户端。对于这种情况，服务端的解决方法是发送之前已经收到的ACK包。也就是说，服务端不会对乱序的包进行确认，而是继续发送对先前正确接收的所有数据包的累积确认ACK。客户端在接收到重复的ACK包时，不会做出任何操作，因为这些ACK包并没有提供新的信息。

在客户端丢包的情况下，丢失的数据包的定时器会超时，此时客户端会重新发送丢失的数据包，即进行超时重传。当丢失的数据包被成功重传并到达服务端时，服务端会收到这个期望的数据包并发送一个新的累积确认ACK包，表示它已成功接收到包括该重传数据包在内的所有数据包。此时，客户端在接收到这个新的ACK包后，会滑动自己的滑动窗口，继续发送下一个数据包。通过这种机制，客户端能够确保丢失的数据包最终被重新发送和正确接收，同时保证了数据的顺序和完整性。



服务端丢包

第二种情况就是服务端丢包。因为服务端窗口大小是1，所以这时候服务端是已经把收到的数据写入了文件中并且使用累积确认机制。所以客户端会收到一个比当前基序号大的ACK包。这时候客户端并不需要像3-1中那样超时重传，因为收到这个ACK包就意味着服务端已经成功接收到了数据包，所以客户端根据ACK包中的acknum更新自己的基序号即可。



检验错误

最后一种情况是数据一部分丢失。若数据传输过程中某些数据包丢失，服务端在收到数据包时会进行检验和校验，以确保数据的完整性。如果服务端发现接收到的数据包的检验和不正确，说明该数据包在传输过程中遭遇了损坏或丢失的情况。这时，服务端不会接受这个数据包，也不会发送ACK确认包，而是直接丢弃该数据包。后续接收数据包过程中由于服务端收到了乱序数据包，所以会重发ACK包，这时候客户端就知道了自己某个包出错了。之后就会进行超时重传。服务端在收到重新发送的正确数据包后，会进行正常的确认和处理。

二、程序设计

本次实验主要改动的地方是文件传输部分，其他代码跟实验3-1基本一样，所以在此就不再进行解释。

首先是客户端的滑动窗口实现。我是这样实现的：首先我维护一个发送队列，所有要发送的数据包都已经保存在这个队列里面，之后在主线程函数中使用互斥锁来保护该队列，并使用条件变量sendCv来保证当前要发送的数据包在滑动窗口里面。而滑动窗口的核心在于维护一个基序号（`base`）和下一个待发送数据包的序列号（`nextseqno`），并动态调整窗口的大小和范围。

在 `canSend()` 函数中，我对当前待发送数据包的序列号（`nextseqno`）与基序号（`base`）之间的差值进行判断，如果差值小于滑动窗口大小的话就代表着能发送当前数据包，否则不能。

为实现实时接收ACK、调整滑动窗口以及处理数据包的重传，我创建了两个专门的线程：`receive_ack` 和 `handle_resend`，在后面会详细解释。

1. `receiveack`：负责监听服务器返回的ACK消息，并根据ACK更新基序号（`base`），从而动态调整发送窗口。
2. `handleresend`：负责检查超时数据包并触发相应的重传操作。

当所有数据包都已发送且接收到对应的ACK时：停止发送线程，通过 `break` 退出主循环。之后等待 `receive_ack` 线程处理完所有ACK，确保资源被正确释放。之后通知重传线程停止，避免继续执行无意义的重传逻辑。

```
// send_data_with_window - 该函数实现了基于滑动窗口的UDP数据包发送机制
// 通过多线程处理接收确认信息，确保数据包的有序发送和超时重传。

bool canSend() {
    return nextseqno - base < win_size;
}

nextseqno = 0;
thread receiveThread(receiveack); //ACK接受线程
thread timeThread(handelresend);  //重传线程

// 循环发送数据包，直到所有数据包发送完毕
while (nextseqno < datanum) {
    // 获取互斥锁，保护共享资源
    unique_lock<std::mutex> lock(sendl);

    // 等待条件变量触发，确保当前可以发送数据包
    sendCv.wait(lock, [] { return cansend(); });

    // 获取当前序列号对应的分组，并设置校验和
    struct data& currentPacket = window[nextseqno];
    setChecksum(&currentPacket);

    // 为当前分组设置定时器的起始时间，用于后续重传或超时处理
    packetTimers[nextseqno] = std::chrono::steady_clock::now();

    // 模拟丢包逻辑：每发送10个数据包，故意丢弃一个
    if (coun == 10) {
        coun = 0; // 重置计数器
        cout << "丢包 seqnum=" << currentPacket.seqnum << endl; // 打印丢包日志
    } else {
        coun++; // 增加计数器
        cout << "发送 seqnum=" << currentPacket.seqnum << endl; // 打印发送日志

        // 使用 sendto 函数发送当前分组到接收方
        sendto(clientSocket, (char*)&currentPacket, sizeof(sendData), 0,
               (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    }
```

```
    // 更新下一个需要发送的序列号
    nextseqno++;

    // 检查是否已经发送完所有数据包，如果是则退出循环
    if (nextseqno >= datanum) {
        break;
    }
}

receiveThread.join(); //等待线程结束
timeThread.join(); //等待线程结束
```

接下来是客户端接收线程的实现。在该线程中，我首先检查服务端返回的累积确认序号是否大于当前的基序号（`base`）。如果确认序号大于 `base`，说明客户端可以接受该确认，并且该 ACK 包是有效的；如果确认序号小于或等于 `base`，则意味着这是一个重复的 ACK 包，客户端记录他的序号，通知重传线程。接收到有效的 ACK 包后，我会停止对应数据包的定时器，并更新基序号。如果此时基序号已经大于或等于要发送的总数据包数（`datanum`），这表明所有数据包都已成功发送并确认，接收线程可以退出并结束。而如果基序号仍然小于数据包总数，则需要通过通知机制唤醒被阻塞的发送线程，允许它继续发送新的数据包。

```
// receiveack - 该函数在一个单独的线程中运行，负责接收来自服务器的ACK确认。
// 它根据接收到的ACK更新基序号（base），并控制滑动窗口的移动，确保可靠的数据包传输。

// 该函数专门用于接收服务器返回的 ACK 消息，并根据 ACK 更新基序号（base）
void receiveack() {
    while (true) {
        // 清空接收缓冲区，准备接收新的数据包
        memset(&receiveData, 0, sizeof(sendData));

        // 接收来自服务器的 ACK 消息
        recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
                 (SOCKADDR*)&servAddr, &addrlen);

        // 如果收到的 ACK 表示成功，并且 ACK 序号大于当前的基序号（base），则更新 base
        if (receiveData.ack == 1 && receiveData.acknum > base) {
            // 获取互斥锁，保证对共享变量 base 的修改是线程安全的
            sendl.lock();

            // 更新基序号（base），滑动窗口向前移动
            base=receiveData.acknum;

            // 打印收到的累积 ACK 序号
            cout << "收到累积 ACK：acknum=" << receiveData.acknum << endl;

            // 如果所有数据包都已确认（base >= datanum），则停止定时器并退出函数
            if (base >= datanum) {
                sendl.unlock();
                stoptimer = true;  // 设置停止定时器标志
                return;  // 结束接收 ACK 线程
            }
            else {
                // 否则，移除已经确认的数据包定时器
```

```
                for (int i = 0; i < base; i++) {
                    if (packetTimers.find(i) != packetTimers.end()) {
                        packetTimers.erase(i);
                    }
                }

            }

            // 释放互斥锁，允许其他线程访问共享资源
            sendl.unlock();

            // 通知发送线程，当前可以发送下一个数据包
            sendCv.notify_one();

            // 重置超时标志，表明没有超时
            timeout.lock();
            timeoutid = -1;  // 清除超时 ID
            timeout.unlock();
        }
        // 如果收到重复的 ACK 或 ACK 序号没有超过当前 base，则说明该 ACK 已经被处理
        else {
            // 打印收到的重复累积 ACK 序号
            cout << "收到重复累积 ACK：acknum=" << receiveData.acknum << endl;

            // 设置超时 ID 为当前的 base，表示当前数据包超时，等待重传
            timeout.lock();
            timeoutid = base;
            timeout.unlock();
        }
    }
}
```

接下来是我的重传线程实现。在该线程中，首先检查是否有需要检测的超时数据包（`timeoutid != -1`）。如果有，则获取当前时间并与该数据包的定时器时间进行对比，检查是否已经超时。如果超时了，说明数据包可能丢失或未被确认，那么我们会重新发送当前滑动窗口内的所有数据包（从 `timeoutid` 开始，直到滑动窗口的大小 `win_size` 结束）。每发送一个数据包后，都会更新该数据包的定时器起始时间，确保超时重传逻辑在下一次检查时能够及时触发。

重传过程会不断重复，直到所有需要重传的数据包被重新发送并成功确认或达到停止条件。为了避免重复重传多遍，所以线程在每次循环后会进行短暂的休眠（`Sleep(100)`）。

```
// 用于处理数据包的重传逻辑
void handelresend() {
    // 循环运行，直到发送任务完成（stoptimer 为 true）
    while (!stoptimer) {
        // 检查是否存在需要处理的超时数据包（timeoutid != -1 表示存在超时包）
        if (timeoutid != -1) {
            // 获取超时队列的锁，确保对 timeoutid 和定时器的操作是线程安全的
            timeout.lock();

            // 获取当前时间
            auto now = std::chrono::steady_clock::now();

            // 检查超时条件：当前时间与定时器起始时间的差值是否超过设定的超时时间
```

```
            if (std::chrono::duration_cast<std::chrono::milliseconds>(now -
packetTimers[timeoutid]).count() > TIMEOUT) {
                // 遍历当前窗口范围内的数据包（从 timeoutid 开始）
                for (int i = timeoutid; i < timeoutid + win_size; i++) {
                    // 如果序号在有效范围内（避免超出数据包总数 datanum）
                    if (i < datanum) {
                        // 打印重传日志
                        cout << "重传 seqnum=" << window[i].seqnum << endl;

                        // 使用 sendto 重新发送当前数据包
                        sendto(clientSocket, (char*)&window[i],
sizeof(sendData), 0,
                               (SOCKADDR*)&servAddr, sizeof(SOCKADDR));

                        // 更新当前数据包的定时器起始时间
                        packetTimers[i] = std::chrono::steady_clock::now();
                    }
                }
            }

            // 释放超时队列的锁
            timeout.unlock();
        }

        // 避免重复重传多遍
        Sleep(100);
    }

    // 发送任务完成，退出函数
    return;
}
```

最后是服务端的累计确认机制。这个实现并不难，在服务端维护一个变量，用它来记录当前已经确认过的序号累积，每次到达一个期望的数据包就更新一次。而当不是期望的数据包的时候服务端重传之前的ACK包即可。

```
// process_received_data - 该函数用于处理接收到的数据包，根据其序列号判断是否按序接收，并进行相应的ACK回复。
// 如果数据包是按序的，则将其写入文件并发送累积确认；如果是乱序的，则丢弃并发送确认；如果数据包无效，则等待重传。

void process_received_data() {
    // 检查接收到的数据包是否有效
    if (checkReceivedData(receiveData)) {
        // 如果数据包的序列号与期望的下一个ACK序列号相同，表示按序接收
        if (receiveData.seqnum == nextAckNum) {
            // 按序数据，直接写入文件
            fwrite(receiveData.data, receiveData.dataLen, 1, p);
            cout << "接收到按序数据包，序列号: " << receiveData.seqnum << ", 数据大小:
"
                 << receiveData.dataLen << " 字节" << endl;

            // 更新期望的下一个序列号
```

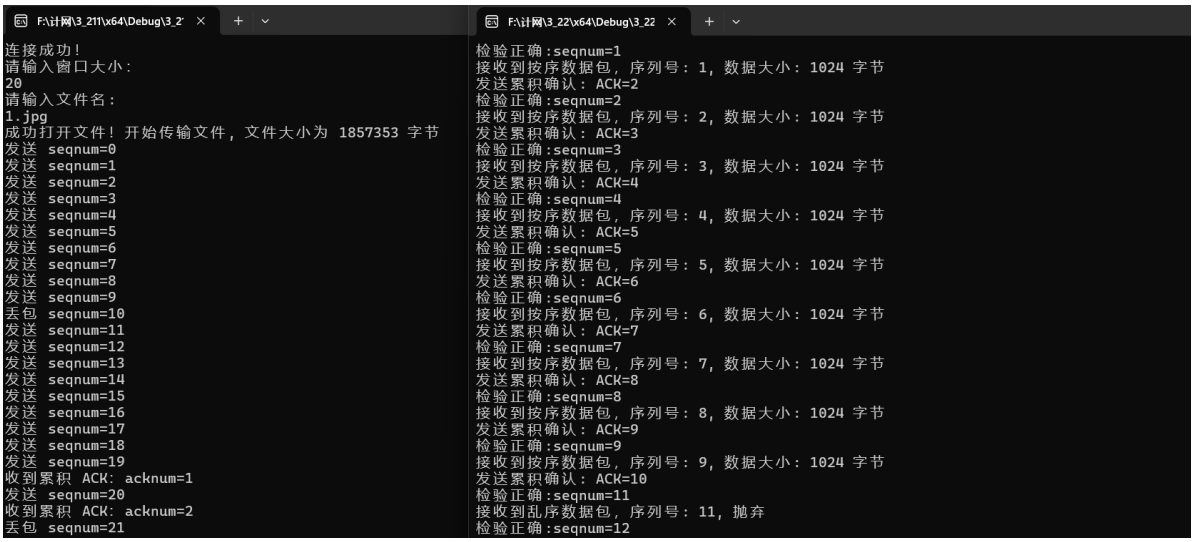
```
            nextAckNum++;

            // 发送累积 ACK，确认已收到到 `nextAckNum - 1` 的数据
            memset(&sendData, 0, sizeof(sendData));  // 清空 sendData 结构
            sendData.ack = 1;  // 设置为 ACK 包
            sendData.acknum = nextAckNum;  // 设置累积确认的序列号
            // 发送累积确认给客户端
            sendto(serverSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&sourceAddr, addrlen);
            cout << "发送累积确认: ACK=" << sendData.acknum << endl;
        }
        // 如果接收到的序列号大于期望的序列号，表示乱序到达
        else if (receiveData.seqnum > nextAckNum) {
            // 乱序数据包丢弃，不处理内容
            cout << "接收到乱序数据包，序列号: " << receiveData.seqnum << ", 抛弃" <<
endl;

            // 发送当前的ACK，以确认接收的序列号（防止客户端重传已收到的数据包）
            sendto(serverSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&sourceAddr, addrlen);
        }
        else {
            // 如果接收到的序列号小于期望序列号，说明该数据包已经接收过或丢失，忽略不处理
            ;
        }
    }
    else {
        // 如果接收到的数据无效，等待超时重传
    }
}
```

三、实验结果

下面我将以1.jpg，滑动窗口为20为例，运行程序。



一开始，客户端会把滑动窗口内的20个数据包全传过去并进入等待状态。因为我设置的丢包率是10%，也就是每10个包丢一次，所以可以看到第10个包是模拟丢包了。而在服务端他在接到一个按序的数据包会发送累积确认ACK，客户端在接到一个期望的累积确认之后滑动base，接着传新的数据包。一直到第10号数据包，服务端会接收到一个乱序的数据包11，这时候服务端会重传之前的累积确认ACK。这时候

客户端会收到大量的重复ACK包。

```
发送 seqnum=27
收到累积 ACK: acknum=9
发送 seqnum=28
收到累积 ACK: acknum=10
发送 seqnum=29
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
收到重复累积 ACK: acknum=10
```

这时候客户端会等待第10号包超时，待超时就要重传10-29这20个目前滑动窗口内的数据包。

```
重传 seqnum=10
重传 seqnum=11
重传 seqnum=收到累积 ACK: acknum=11
12发送 seqnum=30

重传 seqnum=13
重传 seqnum=14
重传 seqnum=15
重传 seqnum=16
重传 seqnum=17
重传 seqnum=18
重传 seqnum=19
重传 seqnum=20
重传 seqnum=21
重传 seqnum=22
重传 seqnum=23
重传 seqnum=24
重传 seqnum=25
重传 seqnum=26
重传 seqnum=27
重传 seqnum=28
重传 seqnum=29
```

后面的重传都是上述这种情况。

最后断开连接，输出统计信息即可。

```
第一次挥手: 客户端发送 FIN 包
第二次挥手: 收到服务器的 ACK, acknum=1815
第三次挥手: 收到服务器的 FIN, seqnum=1
第四次挥手: 客户端发送 ACK
连接已成功关闭!
文件传输完成!
总传输字节数: 1857353 字节
总耗时: 26.741 秒
平均吞吐率: 69457.1 字节/秒
请按任意键继续. . .
```

四、实验总结

实验难点

本次实验相比较实验3-1，遇到的难点还是比较多的。首先因为不是停等机制，所以我们需要把发送和接收两件事情分开，也就是要在两个线程之中运行，而且因为有重传机制，所以发送和重传也需要分开，这也就是说我们要同时执行三个线程，这时候怎么来正确同步三个线程是最重要的。因为一旦不能正确同步，可能会出现死锁或者重复发送没必要的数据包的情况。

第二个难点是在实验过程中发现，当我将数据包的大小设置得过大时，客户端在连续发送多个数据包后，服务端往往只能接收到最多 5 个数据包。这种问题可能是由于客户端发送数据包的速度过快，导致服务端处理不过来，或者是数据包过大导致 UDP 丢包所致。针对这种情况，我想到两种解决方法：一种是在客户端每次发送数据包之间增加一个延时，降低发送速度；另一种是减小单个数据包的大小。在本次实验中，我选择了第二种方法，通过减小数据包的大小来解决这一问题。

第三个难点是如何实现超时重传。在实验 3-1 中，由于每次只需要处理一个数据包，因此可以使用一个专门的线程来计时并进行重传。然而，在本次实验中，由于可能一次性发送多个数据包，我最初的想法是为每个数据包都设置一个单独的线程来计时。但这种方法最终导致了定时器线程之间的混乱和管理问题。为了解决这个问题，我决定采用一种更加简洁有效的方式：使用一个全局字典来保存每个数据包的发送时间。在接收到 ACK 后，我会将相应的数据包序号从字典中删除。这样，我只需要每次检查字典中还未确认的数据包序号，比较它们的发送时间与当前时间的差值，判断是否超过设定的超时值，从而触发重传。

实验改进

在实验过程中，我发现尽管已经使用了互斥锁和条件变量等同步机制，线程之间的操作仍可能出现混乱的情况，例如由于条件变量的唤醒时机不准确，导致线程等待或执行时序不一致。因此，在后续的实验中，我将重点研究并改进以下几个方面：

1. **优化线程同步机制**
    - 分析现有线程之间的交互逻辑，确保条件变量的触发时机更加准确，避免不必要的等待或误唤醒。
    - 引入更细粒度的锁机制，减少线程间的竞争区域，提高同步效率。
2. **增加线程状态监测**
    - 为每个线程设计一个独立的状态变量，用于监测其当前工作状态（如空闲、繁忙、等待），通过状态信息更精准地协调线程的工作。