基于滑动窗口的流量控制

一、实验目的

停等机制（Stop-and-Wait）是最基础的流量控制协议，虽然可以保证传输的稳定可靠性，但它在网络条件较好的时候传输效率较低，主要是由于每次发送数据后都必须等待确认才能继续发送。为了提高传输效率，本次实验将停等协议改进为滑动窗口协议。

滑动窗口机制（Sliding Window）通过允许发送方在接收到接收端返回的ACK确认之前持续发送多个数据包（窗口大小数量），从而能够显著提高了传输效率。滑动窗口机制广泛应用于TCP协议中，它可以有效地提高网络带宽利用率和数据传输的速度。

二、滑动窗口机制原理

滑动窗口机制允许发送方在未接收到先前数据包的ACK确认时，继续发送窗口数量范围内的其他数据包。窗口控制了可以同时发送的数据包的数量，接收方像先前停等协议一样通过ACK确认哪些数据已经成功接收，发送方根据接收到的ACK向后滑动窗口装填入新的数据包并继续发送数据。

1. 滑动窗口的组成

- **窗口大小（Window Size）**：控制同时允许发送的数据包数目。窗口越大，发送方可以发送的未确认数据越多，吞吐量也越高。
  - 发送端：设置固定窗口大小，但若出现较高延迟或乱序发送时，需重传窗口内所有数据包造成较大的开销。
  - 接受端：设置为1，即一次只能够确认一个数据包。
- **序列号（Sequence Number）**：每个数据包都有一个唯一的序列号，接收方通过ACK来标识已成功接收的数据包。
- **累积确认（Cumulative ACK）**：接收方通过发送确认号来告知发送方其已成功接收的数据包的最大序列号。在本次实验中，由于接收端只设置了一个窗口大小，所以只是一次确认一个数据包，和停等协议一致。

2. 窗口管理

发送方维护一个窗口，窗口的大小由 `WINDOW_SIZE` 常量决定。每次发送数据时，发送方会根据当前窗口中的数据包数量决定是否继续发送，如果窗口已满，发送方必须等待收到ACK确认才能滑动窗口，继续发送。

- **发送方**：发送方根据窗口大小限制发送数据包，并在接收到按序的ACK后向后滑动一个窗口，继续发送下一个数据包。
- **接收方**：接收方会保存和更新一个期待的序列号seq，根据接收到的序列号是否符合预期来决定是否发送ACK，如果接收方正确接收到一个数据包，它会向发送端发送确认，若不是预期序列号，则不做任何回应。（此时发送端没有接收到ack就会发生超时重传）

3. 超时重传机制

发送方在发送数据包后启动定时器，在窗口未满时，此时在接受套接字时采用非阻塞状态，即若没有接收到ack就继续向后发送数据包，如果在指定的时间内未收到ACK确认，它会重传窗口中的所有数据包。最大重传次数由 `MAX_RETRIES` 常量控制，如果超过最大重传次数，传输会终止退出程序。如果窗口已满则不再发送数据包，此时会一直堵塞，直到接收到正确的ack释放窗口。

4. 丢包模拟

为了模拟网络环境中的丢包情况，我们在每次发送数据包时以 `PACKET_LOSS_RATE` 的概率丢弃数据包。这样可以检测到发送方如何应对丢包的情况，并验证超时重传正确性。

三、代码实现(只展示更改部分)

1. 数据发送

在数据发送过程中，发送方在窗口范围内循环读取数据包，并每次发送一个数据包到接收方。在每次发送之前，发送方会对每个数据包计算校验和，并决定是否丢包以模拟网络不稳定。发送包的操作在滑动窗口的控制下进行。

```
// 发送数据包
cout <<"=========================================================="<<endl;
cout << "window size: "<< window.size() << endl;
window.push_back(sendMsg);  // 把数据包放入窗口
if (dis(gen) < PACKET_LOSS_RATE) {
    cout << "[模拟丢包] 未发送 packet " << window[cur].seq <<endl;
} else {
    sendto(sockfd, &window[cur], sizeof(window[cur]), 0,(conststruct sockaddr *)&serveraddr
    cout << "Send packet " << seq << ", size: " << window[cur]len <<" bytes, 校验和: " << wi
}
auto cktime_start = chrono::high_resolution_clock::now();
seq++; // 增加序列号
cur++;

int valread = recvfrom(sockfd, &recvMsg, sizeof(recvMsg)MSG_DONTWAIT, (struct sockaddr *)&se
auto cktime_end = chrono::high_resolution_clock::now();
//收到ack
if (valread > 0 && recvMsg.type == ACK) {
    // 滑动窗口
    cout << "received ack=" << recvMsg.ack << ", sliding windowfornext packet " << endl;
    window.erase(window.begin());
    cur--; // 减少一位
    delay = chrono::duration<double>::zero();
}
```

- 每次发送一个数据包，发送方会先检查窗口的大小。然后将当前的消息包sendMsg加入发送窗口window中。如果触发模拟丢包（即dis(gen) < PACKET_LOSS_RATE），则模拟丢包并不发送该数据包；否则，使用sendto()将数据包发送给接收方。发送的每个数据包都会输出序列号、数据包大小和校验和。
- 滑动窗口机制：每发送一个数据包，序列号seq和窗口指针cur都会增加。在接收到ACK时，窗口会滑动，当前发送的包会从窗口中移除（通过window.erase(window.begin())）。如果ACK确认成功，cur会减少，窗口向前滑动。

2. 窗口维护

当窗口已满时，发送方需要等待接收到ACK确认才能继续发送下一个数据包。在此期间，如果未收到ACK，发送方会进行超时重试，重新发送窗口中的所有数据包，直到收到ACK或达到最大重试次数。

```
// 窗口已满，阻塞直到接收到 ACK
if(window.size() == WINDOW_SIZE) {
    cout << "窗口已满，阻塞直到接收到 ACK" << endl;
    retries = 0;
    while(retries < MAX_RETRIES) {
        // 检测超时
        struct timeval timeout = {TIMEOUT_MS / 1000, (TIMEOUT_MS %1000) * 1000};
        fd_set fds;
        FD_ZERO(&fds);
        FD_SET(sockfd, &fds);
        if (select(sockfd + 1, &fds, nullptr, nullptr, &timeout) >0) {
            int valread = recvfrom(sockfd, &recvMsg, sizeo(recvMsg), 0, (struct sockaddr *)&
            if (valread > 0 && recvMsg.type == ACK) {
                if (recvMsg.ack == window[0].seq + 1) {
                    // 滑动窗口
                    cout << "received ack=" << recvMsg.ack << ",sliding window for next pacl
                    window.erase(window.begin());
                    cur--; // 减少一位
                }
                break;  // 收到ACK，跳出重试
            }
        }
        retries++;
        cout << "Timeout or incorrect ACK. Retrying... (" << retries<< "/" << MAX_RETRIES <<
        for(int i=0; i<window.size(); i++) {
            sendto(sockfd, &window[i], sizeof(window[i]), 0, (conststruct sockaddr *)&server
            cout << "[重新发送] Send packet " << window[i].seq << ",size: " << window[i].len
        }
    }
    // 超过最大重传次数
    if (retries == MAX_RETRIES) {
        cout << "Failed to receive ACK after " << MAX_RETRIES << "retries. Give up transfer
        break;
    }
}
```

若发送端数据包全部发送完毕，则会开始一直循环接收ack，直到全部正确按序接收到所有ack后才算完成所有传输，所以这里又设置了一个发送完毕的状态使程序进入，防止在还没有接受完成就退出程序。在最后发送END类别序列包表示结束传输，接收端收到后也退出程序。

```
// 全部发送完毕，等待剩余ack
    if(sendMsg.len == 0){
        cout<< "全部发送完毕，等待剩余ack..."<< endl;
        while(window.size()!=0){
            while(retries < MAX_RETRIES) {
                // 检测超时
                struct timeval timeout = {TIMEOUT_MS / 1000, (TIMEOUT_MS % 1000) * 1000};
                fd_set fds;
                FD_ZERO(&fds);
                FD_SET(sockfd, &fds);
                if (select(sockfd + 1, &fds, nullptr, nullptr, &timeout) > 0) {
                    int valread = recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr
                    if (valread > 0 && recvMsg.type == ACK) {
                        if (recvMsg.ack == window[0].seq + 1) {
                            // 滑动窗口
                            cout << "received ack=" << recvMsg.ack << ", sliding window for next
                            window.erase(window.begin());
                            cur--; // 减少一位
                        }
                        break;  // 收到ACK，跳出重试
                    }
                }
                retries++;
                cout << "Timeout or incorrect ACK. Retrying... (" << retries << "/" << MAX_RETRIE
                for(int i=0; i<window.size(); i++) {
                    sendto(sockfd, &window[i], sizeof(window[i]), 0, (const struct sockaddr *)&se
                    cout << "[重新发送] Send packet " << window[i].seq << ", size: " << window[i]
                }
            }
            // 超过最大重传次数
            if (retries == MAX_RETRIES) {
                cout << "Failed to receive ACK after " << MAX_RETRIES << " retries. Give up trans
                return;
            }
        }
        //全部接收到ack，发送end终止
        sendMsg.type = END;
        sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const struct sockaddr *)&serveraddr, server
        cout << "文件传输完成！" << endl;
        break;
    }
```

3. 接收端接收和确认

```
// 校验校验和
u_long receivedChecksum = recvMsg.checksum;
if (calculateChecksum(recvMsg) != receivedChecksum) {
    cout <<"=======================================================" <<endl;
    cout << "Checksum error for packet " << recvMsg.seq << ".Retry..." << endl;
    continue;
}
if (recvMsg.seq == expectedSeq) {
    cout <<"=======================================================" <<endl;
    cout << "Received packet " << recvMsg.seq << ", size: " <<recvMsg.len << " bytes" << end
    // // 模拟固定延迟
    // std::this_thread::sleep_for(std::chrono::milliseconds(500));
    // 发送 ACK
    sendMsg.type = ACK;
    sendMsg.ack = recvMsg.seq + 1;
    sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const structsockaddr *)&cliaddr, cliaddr_l
    expectedSeq++;
    cout << "校验和: "<< calculateChecksum(recvMsg) << " 校验和正确"<< ", Send ack="<< sendMs

    if (recvMsg.len > 0) {
        output_file.write(recvMsg.data, recvMsg.len);
    }
    continue;
}
```

- 序列号确认：接收方会检查数据包的序列号是否与预期的序列号expectedSeq匹配。如果匹配，接收方表示已成功接收到该数据包。
- 发送ACK：接收方发送ACK给发送方，告知数据包已收到，并附带下一个期望的序列号（即recvMsg.seq + 1）。接收方的expectedSeq会递增，以保证数据包的顺序。

4. 超时重传

如果发送方在规定时间内没有收到ACK确认，重传窗口中的所有数据包。通过使用 `select` 函数来等待数据包的ACK，超时后进行重传。
如果重试次数达到最大值（MAX_RETRIES），发送方会放弃当前的数据传输。

```
while(retries < MAX_RETRIES){
    // 超时重传窗口内所有包
    cout << "Timeout or incorrect ACK. Retrying... (" << retries << "" << MAX_RETRIES << ")"
    for(int i=0; i<window.size(); i++) {
        sendto(sockfd, &window[i], sizeof(window[i]), 0, (conststruct sockaddr *)&serveraddr,
        cout << "[重新发送] Send packet " << window[i].seq << ",size: " << window[i].len << "
    }
    int valread = recvfrom(sockfd, &recvMsg, sizeof(recvMsg),MSG_DONTWAIT, (struct sockaddr *
    if (valread > 0) {
        if (recvMsg.type == ACK && recvMsg.ack == window[0].seq + 1) {
            // 接收到延迟的ack
            cout << "received ack=" << recvMsg.ack << ", continue tosend next packet" << endl
            window.erase(window.begin());
            cur--;
            delay = chrono::duration<double>::zero();
            break;
        }
    }
    retries++;
}
// 超过最大重传次数
if (retries == MAX_RETRIES) {
    cout << "Failed to receive ACK after " << MAX_RETRIES << "retries. Give up transfer! " <<
    break;
}
```

5. 累计确认

发送端收到的ack大于当前要接收的ack,即 `recvMsg.ack >= window[0].seq + 1` ，则会将所收到ack前的所有数据包全部确认，窗口移动 `recvMsg.ack - window[0].seq` 个数量，实现累计确认。若接收端在返回ack时出现丢包，没有连续的返回所有正确ack，也可以同样将最后一个ack前的所有数据包作为已确认。

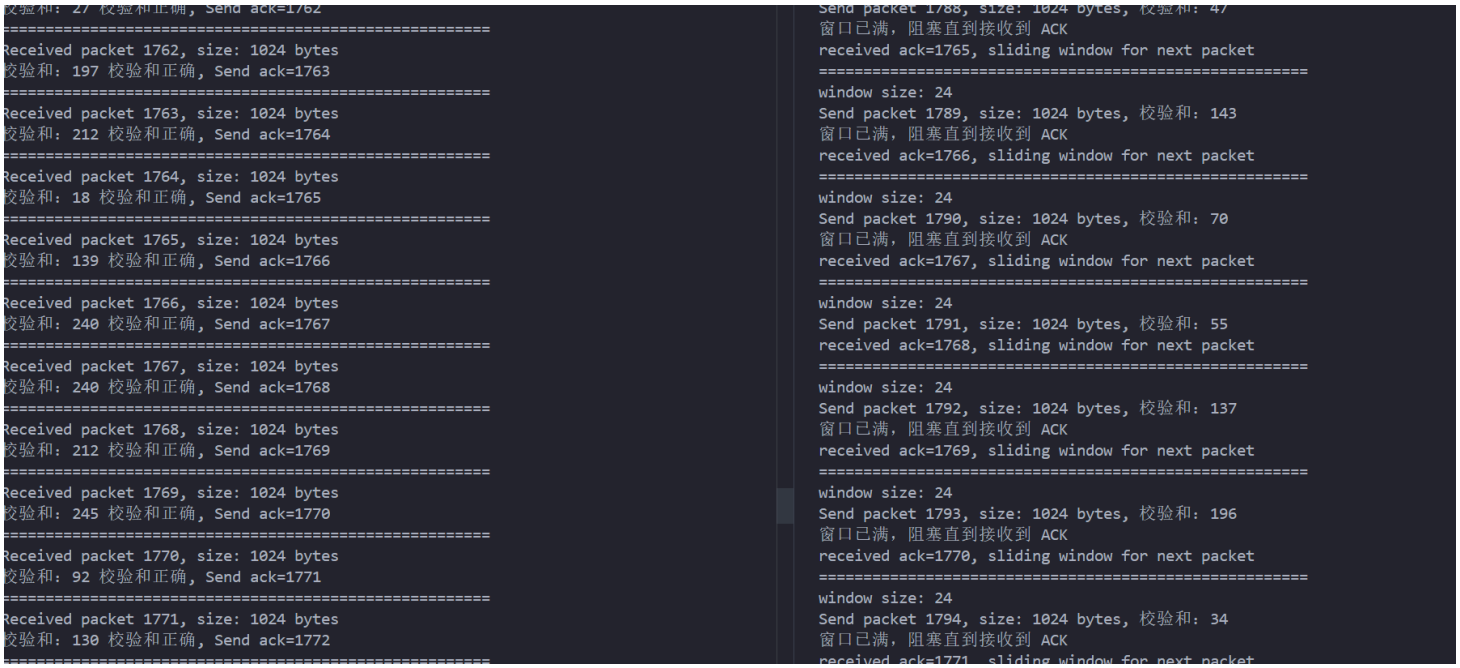
```
if (valread > 0) {
    if (recvMsg.type == ACK && recvMsg.ack >= window[0].seq + 1) {
        // 接收到延迟的ack
        cout << "received ack=" << recvMsg.ack << ", continue to send next packet" << endl;
        int slide = recvMsg.ack - window[0].seq;
        for(int i=0; i<slide; i++){
            window.erase(window.begin());
            cur--;
        }
        delay = chrono::duration<double>::zero();
        break;
    }
}
```

四、测试与分析

设置窗口大小为25，超时时间限制1000ms

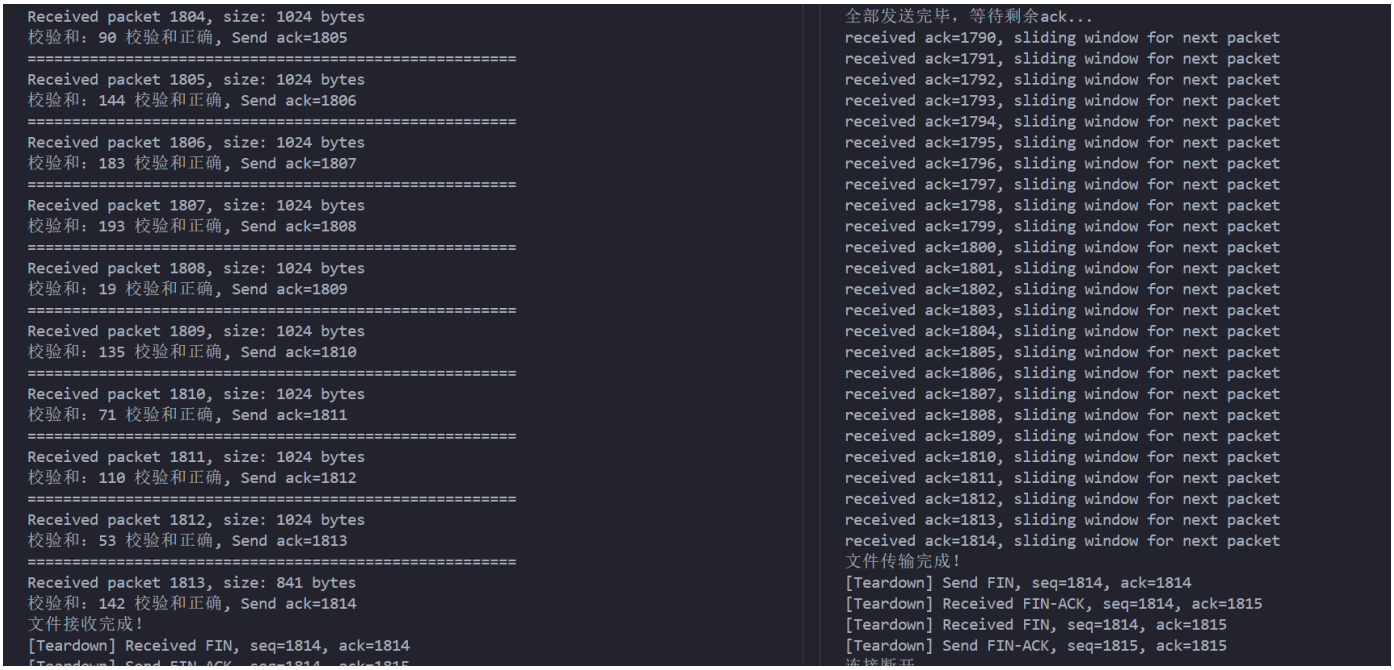
```
// 定义窗口大小
#define WINDOW_SIZE 25

#define TIMEOUT_MS 1000 // 超时时间（毫秒）
```

1. 正常传输（没有设置丢包程序）



- 没有等待接收端发来ack，提前在窗口大小范围内发送

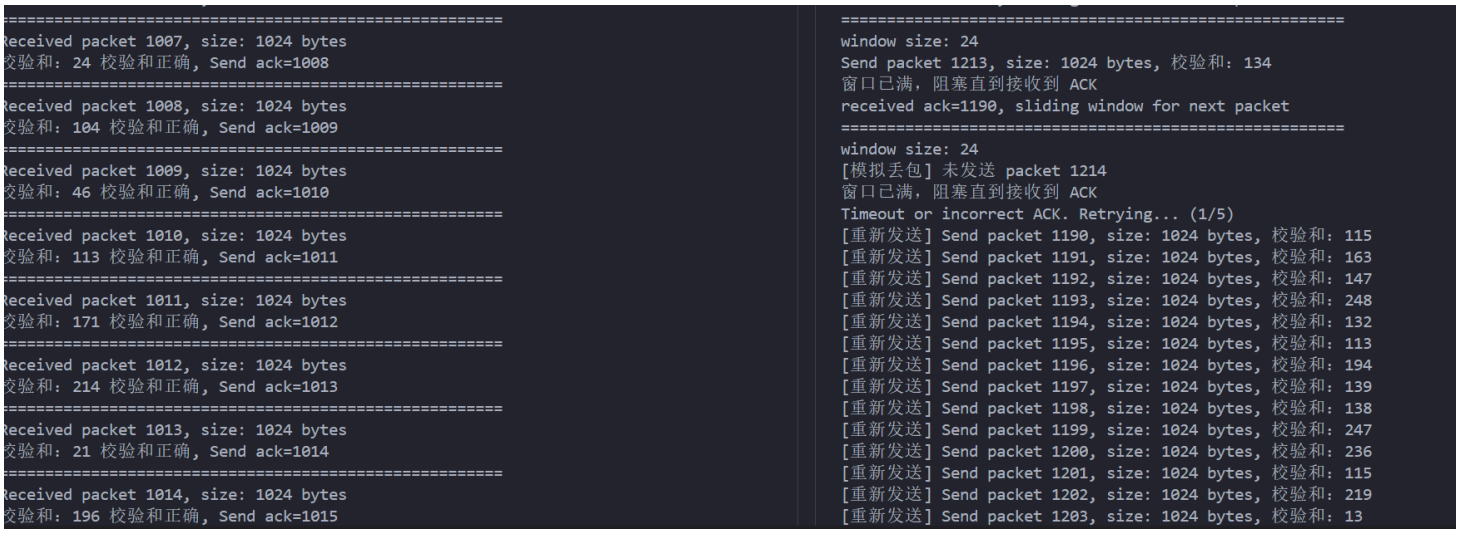
- 当窗口已满时进行阻塞，停止发送数据包，直到接收到正确顺序的ack



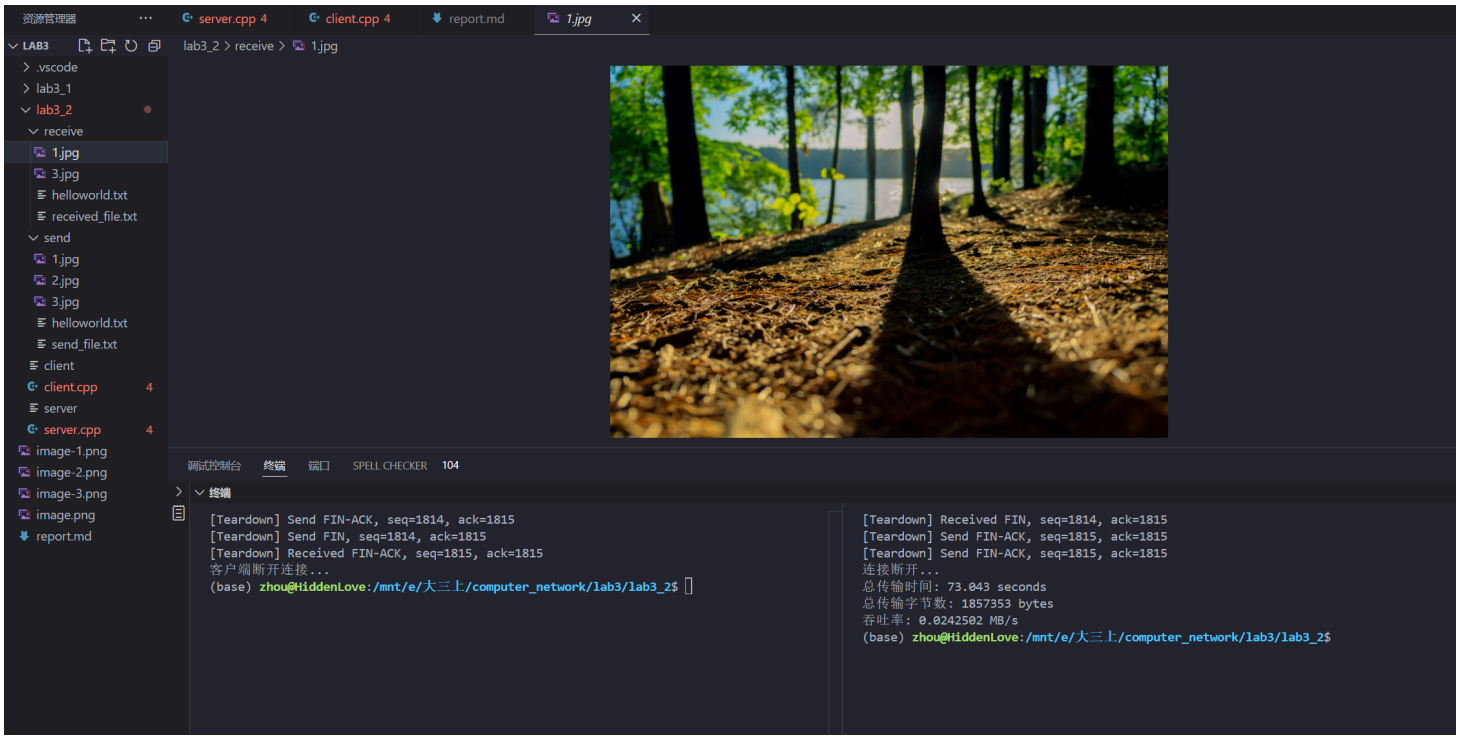
- 在全部发送完毕后没有立刻停止，而是继续接受剩余的ack，全部接收完毕代表所有数据包达到，结束传输。

2. 设置丢包率，验证超时重传

```
#define PACKET_LOSS_RATE 0.5 // 丢包率
```



- 设置丢包率，实现随机丢包
- 由于丢包后，接收端并没有接收到期望的序列号，因此没有返回ack，发送端因此需要重新传输窗口内的所有包来保证传输正确。



最终图片数据正确传输

五、总结

滑动窗口协议相较于停等协议提供了更高的吞吐量、更低的延迟和更高效的带宽利用，适合高延迟和高带宽的网络环境。

然而，同时也存在一些缺点，它的实现复杂度高，需要更多的内存和缓存支持，并且在网络状况不稳定时需要重新传输窗口内所有数据包，可能导致接收缓冲崩溃等不稳定的情况。（代码有时会因为超过重传次数导致传输失败）。

后续进行改进：

- 根据网络状况动态调整窗口大小，进一步提高吞吐量。
- 结合的拥塞控制算法（如慢启动、拥塞避免等），优化流量控制。