

Lab3-2

窗口状态变化日志信息补充

可以看到，虽然窗口的大小是固定的，但是在刚开始发送报文的时候，窗口内缓存的数据报文的数量是不断上升的。

然后当达到16的时候，就会等待接收成功然后窗口前移。如图就是在倒数第三个 [WIN-STATE] 日志中，

可以看到size减少了2，这是因为ACK报文的中确认号确认了两个报文，所有缓冲区缩小2，窗口前移两个单位。

后续可以看到窗口内的缓存的报文数量又开始上涨，nextSeq的值也在不断增加。

```
运行: server_3_2
[WIN-STATE]: { [size:9] [begin:1] [end:9] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:8] [Len:1024] }
[Seg 7 in 1814]
[WIN-STATE]: { [size:10] [begin:1] [end:10] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:9] [Len:1024] }
[Seg 8 in 1814]
[WIN-STATE]: { [size:11] [begin:1] [end:11] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:10] [Len:1024] }
[Seg 9 in 1814]
[WIN-STATE]: { [size:12] [begin:1] [end:12] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:11] [Len:1024] }
[Seg 10 in 1814]
[WIN-STATE]: { [size:13] [begin:1] [end:13] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:12] [Len:1024] }
[Seg 11 in 1814]
[WIN-STATE]: { [size:14] [begin:1] [end:14] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:13] [Len:1024] }
[Seg 12 in 1814]
[WIN-STATE]: { [size:15] [begin:1] [end:15] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:14] [Len:1024] }
[Seg 13 in 1814]
[WIN-STATE]: { [size:16] [begin:1] [end:16] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:15] [Len:1024] }
[Seg 14 in 1814]
[WIN-STATE]: { [size:16] [begin:1] [end:16] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:16] [Len:1024] }
[Seg 15 in 1814]
[RECV]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK]: Package (SEQ to : 3) sent successfully!
[WIN-STATE]: { [size:14] [begin:3] [end:17] }
[WIN-STATE]: { [size:15] [begin:3] [end:17] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:17] [Len:1024] }
[Seg 16 in 1814]
[WIN-STATE]: { [size:16] [begin:3] [end:18] }
[SEND]: { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:18] [Len:1024] }
[Seg 17 in 1814]
```

1. 问题描述

在上一次实验中，已经利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传等。

本次实验在上一次实验的基础上继续进行改进，实现了基于GBN的滑动窗口流量控制机制，并且在接收端选择采用累计确认的实现方式。

基于给定的实验要求，在实验中，由于实现的是单向的通信，即只从服务器发送给客户端，服务器作为发送端，客户端作为接收端。

因此在建立连接和断开连接的过程中，不适合使用TCP的三次握手和四次挥手，在实验中对建立连接和断开连接的过程进行了简化。

建立连接和断开连接的过程采用了两次握手和两次挥手的方式。

在数据传输的过程中，采用了GBN流水线协议进行实现。

在流量控制方面，基于GBN滑动窗口机制，使用固定窗口大小，实现流量控制。

由于采用流水线机制，如果采用一次一应答的方式，一旦产生丢包，会导致收到大量重复ACK，

这会使得资源严重浪费。

因此在本次实验的实现中，采用了累计确认的方式，即只有累计收到了N条有效报文之后，才会回应一个最大连续序列号的ACK。

2. 报文设计

在本次实验中，对于数据报文的格式并没有进行调整，仍采用上一次实验的数据报文格式。其中包含了发送端和接收端的端口号，确认号，序列号，标志位，长度，校验和以及数据段，整体按照16位进行对齐。

```
1 struct Message {
2     unsigned short sourcePort{};
3     unsigned short destinationPort{};
4     int ack{};
5     int seq{};
6     unsigned short flagAndLength = 0;
7     unsigned short checksum{};
8     char data[MSS]{};
9 };
```

其中对于标志位的设计如下，由于各个数据段要按照16位对齐，因此将标志位和长度合并到同一个16位中，其中高五位为标志位，低12位表示数据段的长度。

1		15		14		13		12		11		10-0	
2		REP		FIN		SYN		ACK		FHD		LEN	

- 第15位表示当前的数据包是否为重传数据包
- 第14位表示FIN标志位
- 第13位表示SYN标志位
- 第12位表示ACK标志位
- 第11位表示当前的数据包的数据段是否为文件头信息
- 第10-0位表示数据段的长度

对于伪首部，同样参考了TCP的伪首部设计，其中包含了发送端和接收端的IP地址，zero，协议号，以及数据报的长度。

```
1 struct PseudoHeader {
2     unsigned long sourceIP{};
3     unsigned long destinationIP{};
4     char zero = 0;
5     char protocol = 6;
6     short int length = sizeof(struct Message);
7 };
```

基于GBN滑动窗口的流量控制

在本次实验中，基于GBN流水线协议，使用固定窗口大小，实现了流量控制机制。

首先分析一下发送端的动作，考虑窗口大小为N的情况，发送端的发送缓冲区大小和窗口大小一致，也就是发送端最多一次性发送N个报文。

通过两个指针来控制当前已经发送的报文序号和还未确认的报文序号，如下图所示。

base指针表示的是当前已经确认的最大序列号，nextseq指针表示的是当前已经发送的最大序列号，这之间的报文就是发送了但是还未被确认的部分。

每当发送端接收到回应的ACK的时候，需要判断一下当前回应的序号是否超过了base，如果超过了base，则说明从base到回应序号这部分的报文已经被确认了。

此时可以向前滑动窗口，调整base指针，并且重新启动定时器。

而如果在定时器超时时间内都没有收到有效的ACK，则在定时器超时之后，发送端会重新发送从base到nextseq之间的全部报文。

分析接收端的动作，在本次实验中，由于采用了GBN流水线协议，因此可以将接收端缓冲区简化为大小只为1的队列。

每次收到发送端发送来的报文的时候，判断当前的序号是否为自己期待的序号，如果相符则接收成功，如果不相符，则返回已经确认的最大序号。

此时，如果发送端发送分组的第二个报文丢失，则整个分组都需要回复第一个序号，由此会导致出现大量重复序号的ACK，这回极大浪费链路资源。

因此在本次实验中，对于接收端并没有采用一次一应答的方式，而是采用了累计确认的方式，即等待收到指定K个有效报文之后，再统一回应依次最大确认序号。

如果没有收到有效数量的报文，并不会导致两端卡死，因为发送端有超时重传机制，因此当长时间未收到有效ACK的时候，会超时重传。



校验和

为了保证数据在传输过程中没有发生错误，或者是检查数据在传说过程中是否发生了错误，在实验中仿照UDP的校验和完成了差错检验。

这里的实现原理同课内理论完全一致，就不再赘述，直接上代码。

```
1 void Message::setChecksum(struct PseudoHeader *pseudoHeader) {
2     this->checksum = 0;
3     unsigned long long int sum = 0;
4     for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
5         sum += ((unsigned short int *) pseudoHeader)[i];
6     }
7     for (int i = 0; i < sizeof(struct Message) / 2; i++) {
8         sum += ((unsigned short int *) this)[i];
9     }
10    while (sum >> 16) {
11        sum = (sum & 0xffff) + (sum >> 16);
12    }
13    this->checksum = ~sum;
14 }
```

```

15
16 bool Message::checksumValid(struct PseudoHeader *pseudoHeader){
17     unsigned long long int sum = 0;
18     for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
19         sum += ((unsigned short int *) pseudoHeader)[i];
20     }
21     for (int i = 0; i < sizeof(struct Message) / 2; i++) {
22         sum += ((unsigned short int *) this)[i];
23     }
24     while (sum >> 16) {
25         sum = (sum & 0xffff) + (sum >> 16);
26     }
27     return sum == 0x000ffff;
28 }

```

超时重传

在本次实验中，由于每次收到有效ACK之后，都会重新启动超时器，因此继续采用上一次实验中的超时机制就显得不太合适。

为了能够更加方便地实现超时重传的机制，在本次实验中，封装了一个计时器类，其中提供了启动计时器，停止计时器的功能。

配合上发送端的超时重传线程，可以实现超时重传的功能。

计时器类的具体实现如下，主要提供了 `start()` 函数和 `stop()` 函数，分别进行计时器的启动和停止。

此外还提供了超时判断函数 `isTimeout()`，用于检测当前是否超时。

```

1  class Timer {
2      bool isStart;
3      int timeCnt = 0;
4      int timeOut = RTO;
5  public:
6      Timer() {
7          isStart = false;
8      }
9      void start() {
10         isStart = true;
11         timeCnt = 0;
12     }
13     void stop() {
14         isStart = false;
15     }
16     bool isTimeout() {
17         if (isStart) {
18             sleep(1);
19             timeCnt++;
20             if (timeCnt >= timeOut) {
21                 timeCnt = 0;
22                 return true;
23             }
24         }

```

```

25         return false;
26     }
27 };

```

在发送端，专门使用一条线程来解决超时重传的工作。这条线程负责检查当前是否触发了超时机制。

如果触发了超时机制，则需要将发送缓冲区内未被确认的报文全部重新发送。

这里要注意的事情是，由于需要对发送缓冲区进行操作，而同一时间在发送线程里可能会有向发送缓冲区增加报文的操作。

为了避免冲突，需要给发送缓冲区加锁。

```

1 void beginTimeout() {
2     std::thread resendThread([&]() {
3         while(true) {
4             while(!timer.isTimeout()) {}
5             std::string log = "[TIMEOUT] : Package (SEQ from : " +
std::to_string(baseSeq) + " to : " + std::to_string(nextSeq - 1) + ")
re-send!";
6             logger.addLog(log);
7             int i = baseSeq;
8             do {
9                 std::lock_guard<std::mutex> lockGuard(bufferMutex);
10                struct Message message = sendBuffer[i-baseSeq];
11                sendto(serverSocket, (char*) &message, sizeof(struct
Message), 0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR));
12                logger.addLog("[RE-SEND] : " +
message2string(message));
13            } while(++i < nextSeq);
14        }
15    });
16    resendThread.detach();
17 }

```

发送端

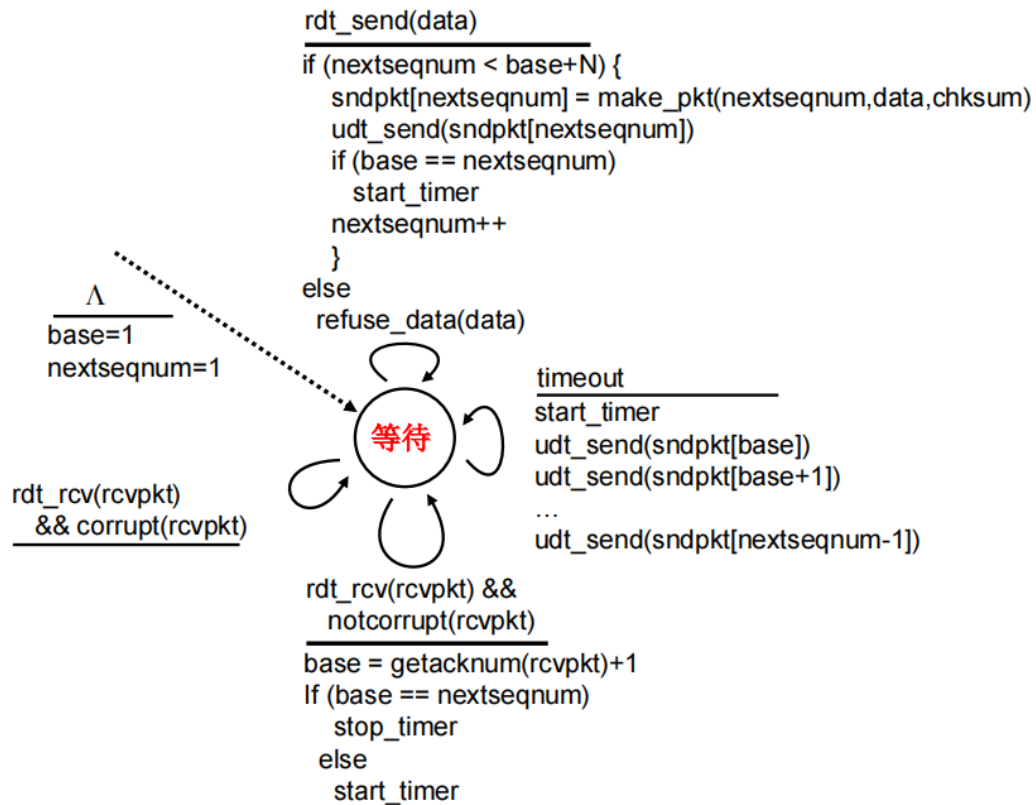
发送端使用了GBN流水线协议，并且采用了固定窗口大小的发送缓冲区。

通过base指针和nextSeq指针分别来描述当前确认的最大报文序号，和当前发送过的最大报文序号。

其整体流程如下图所示。发送端的动作主要包含四个。

1. 发送数据的时候，首先要判断当前的发送缓冲区中是否还有空闲的位置，如果有空闲的位置，才能够进行发送。
这样也就是实现了一个简单的流量控制机制。每成功发送一个新的报文之后，都要将nextSeq向后进行调整，表示当前已经发送了这个报文，并等待确认中。
如果发送缓冲区已经清空了，则需要重新开始计时。
2. 当收到ACK确认报文的时候，需要判断当前确认的报文序号是否大于当前已经确认的最大报文序号，如果是，则需要将base指针向后移动，表示当前已经确认了这个报文。
每收到一个ACK之后，都会重置计时器，重新开始计时，等待超时重传。
3. 当收到了一个损坏的报文的时候，就什么都不做，等待超时重传。

4. 当触发超时机制的时候，超时重传工作线程会启动，将发送缓冲区中未被确认的报文全部重新发送。



在代码实现方面，发送缓冲区直接采用一个双端队列，通过baseSeq和nextSeq来模拟发送缓冲区的情况。

当需要发送报文的时候，会首先判断baseSeq和nextSeq之间的差值是否小于窗口大小WINDOW_SIZE，也就是说发送缓冲区是否还有空闲位置。

如果没有，则进行等待，如果有的话，则给发送缓冲区上锁，将待发送的报文加入到发送缓冲区中，然后发送报文。

```

1 void sendPackage(struct Message message) {
2     // add to the send buffer
3     std::lock_guard<std::mutex> lockGuard(bufferMutex);
4     sendBuffer.push_back(message);
5     // send package
6     if(!randomLoss()){
7         sendto(serverSocket, (char*) &message, sizeof(struct Message),
0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR));
8     }
9     logger.addLog("[SEND] : " + message2string(message));
10    // start timer
11    if(baseSeq == nextSeq) {
12        timer.start();
13    }
14    // update nextSeq
15    nextSeq++;
16 }

```

发送端专门有一条接收线程来接收ACK报文。由于接收端采用的是确认重传，因此并不会每一个报文都回应ACK，而是累计确认收到了一定数量之后再统一回应最大确认序号。因此发送端在接收到ACK之后，当检查报文无误后，需要根据接收端确认的最大序号，从发送缓冲区中将之前确认过的报文全部弹出。而这个过程需要上锁，避免发送线程和接收线程同时操作发送缓冲区导致冲突。接下来就是根据接收到的被确认最大序号，更新baseSeq，也就是实现了窗口的滑动。

```
1 void beginRecv() {
2     std::thread recvThread([&](){
3         while(true) {
4             struct Message recvBuffer;
5             int clientAddressLength = sizeof(SOCKADDR);
6             int recvLength = recvfrom(serverSocket, (char *)
&recvBuffer, sizeof(struct Message), 0, (struct sockaddr *)
&clientAddress, &clientAddressLength);
7             // if receive buffer is valid
8             if(recvLength > 0) {
9                 if(recvBuffer.checksumValid(&recvPseudoHeader)) {
10                     logger.addLog("[RECV] : " +
message2string(recvBuffer));
11                     if(recvBuffer.isSYN() && recvBuffer.isACK()) {
12                         logger.addLog("[LOG] Connection
established!");
13                     }
14                     else if(recvBuffer.isFIN() && recvBuffer.isACK())
{
15                         logger.addLog("[LOG] Connection destroyed!");
16                     }
17                     else {
18                         std::string log = "[ACK] : Package (SEQ to : "
+ std::to_string(recvBuffer.seq) + ") sent successfully!";
19                         logger.addLog(log);
20                     }
21                     // update send buffer
22                     for(int i = baseSeq; i <= recvBuffer.ack; i++) {
23                         std::lock_guard<std::mutex>
lockGuard(bufferMutex);
24                         sendBuffer.pop_front();
25                     }
26                     if(recvBuffer.isFIN()) {
27                         return;
28                     }
29                     // update baseSeq
30                     baseSeq = recvBuffer.ack + 1 > baseSeq ?
recvBuffer.ack + 1 : baseSeq;
31                     // check whether to reset timer
32                     if(baseSeq == nextSeq) {
33                         timer.stop();
34                     }
35                     else {
36                         timer.start();
```



```

37         }
38     }
39     else {
40         logger.addLog("[LOG] checksum invalid! wait for
timeout!");
41     }
42 }
43 else {
44     logger.addLog("[ERROR] : Package received from socket
failed!");
45 }
46 }
47 });
48 recvThread.detach();
49 }

```

接收端

由于采用了GBN流水线协议，为了简化接收端的操作，接收端并没有使用接收缓冲区。只是使用了一个 `EXPECT_SEQ` 来表示当前等待的序号。

接收端的操作如下：当接收到一个有效的报文之后，会检查当前的报文包含的序号和自己期望等待的序号是否一致。

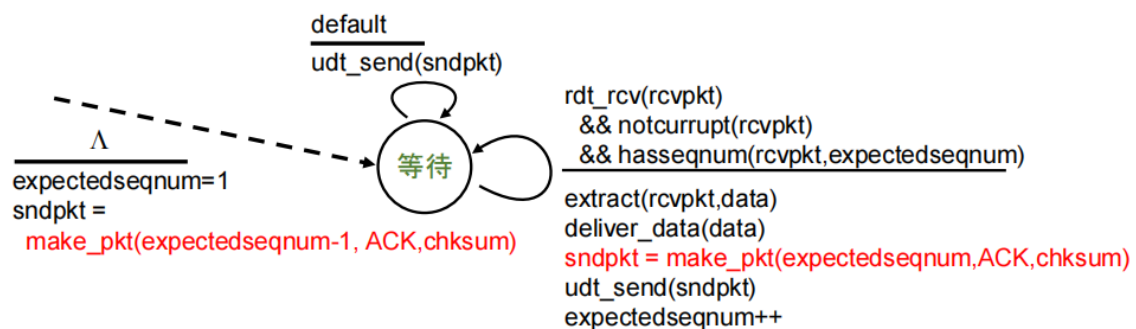
如果一致则证明当前的报文是有效的，将其接收并传给上层应用。此时回应的确认号就是收到的报文的序号，并且将 `EXPECT_SEQ` 向后递增。

而如果报文无效或者序号和期待的序号不一致，则说明发生了失序，在GBN中需要将这个报文丢弃，并且回应确认号为 `EXPECT_SEQ-1`。

而在本次实验中，为了减少接收端回应的重复ACK数量，采用了累计确认的机制。即接收端并不是接收到一个有效报文就回应ACK，而是等待接收到指定K个有效报文之后，才会统一回应一个最大的有效报文序号。

接收端不需要考虑超时重传的问题，因为如果接收端回复的ACK失败了，会触发发送端的超时重传，此时接收到的序列号和期望确认的序列号不符，而回复的确认号正好为接收到数据包的序列号。

因此发送端在接收到这个ACK之后就会进入下一个序列号的发送状态，整个流程就恢复正常了。



具体代码实现如下，这里的代码十分冗长，是因为将这个接受逻辑都封装在这一个函数内部了。

因此接收端接收到数据包之后，需要根据不同的标志位分别进行处理。

- 当接收到的是服务端发送的SYN的时候，表示当前正处于建立连接的阶段，客户端应当回应ACK SYN报文
- 当接收到的是服务端发送的FIN的时候，表示当前正处于断开连接的阶段，客户端应当回应ACK FIN报文，并且进入等待退出的阶段。
拉起一个计时器，等待两倍的RTO，如果两倍的RTO内没有接到服务端重发的FIN则说明能够正常断开，否则的话需要进行重传并重置计时器状态。
- 当接收到的数据段是服务端发送的数据段的时候，需要判断当前传输的数据段包含的内容是文件头还是文件内容
 - 如果当前的数据段包含的是文件头，则根据数据段的描述信息创建文件，然后回复相应的ACK序列号
 - 如果当前的数据段包含的文件内容，则进行写文件的操作，然后回复指定相应的ACK序列号

```

1  [[noreturn]] void beginRecv() {
2      // receive message from server
3      struct Message recvBuffer{};
4      // write to file
5      std::fstream file;
6      unsigned int fileSize;
7      unsigned int currentSize = 0;
8      std::string filename;
9      // cumulative ACK
10     int ackNum = 0;
11     while(true) {
12         int serverAddressLength = sizeof(SOCKADDR);
13         int recvLength = recvfrom(clientSocket, (char*) &recvBuffer,
sizeof(struct Message), 0, (struct sockaddr *) &serverAddress,
&serverAddressLength);
14         if(recvLength > 0) {
15             std::cout << "[RECV] : ";
16             printMessage(recvBuffer);
17             if(recvBuffer.checksumValid(&recvPseudoHeader) &&
recvBuffer.seq == EXPECT_SEQ) {
18                 // update current seq
19                 EXPECT_SEQ += 1;
20                 // update cumulative ACK
21                 ackNum += 1;
22                 // if message is SYN
23                 if(recvBuffer.isSYN()) {
24                     if(!randomLoss()) {
25                         sendACKSYN(recvBuffer.seq);
26                     }
27                 }
28                 // if message is FIN
29                 else if(recvBuffer.isFIN()) {
30                     if(!randomLoss()) {
31                         sendACKFIN(recvBuffer.seq);
32                     }
33                     exitTime = 0;

```

```

34         waitExit();
35     }
36     // if message contains data
37     else {
38         // if message contains file header
39         if(recvBuffer.isFHD()) {
40             struct FileDescriptor fileDescriptor{};
41             memcpy(&fileDescriptor, recvBuffer.data,
sizeof(struct FileDescriptor));
42             std::cout << "Receive file header: [Name:" <<
fileDescriptor.fileName << "] [Size:"
43                 << fileDescriptor.fileSize << "]" <<
std::endl;
44             fileSize = fileDescriptor.fileSize;
45             filename = fileDir + "/" +
fileDescriptor.fileName;
46             currentSize = 0;
47             // create file
48             file.open(filename, std::ios::out |
std::ios::binary);
49         }
50         else {
51             // write to file
52             file.write(recvBuffer.data,
recvBuffer.getLen());
53             currentSize += recvBuffer.getLen();
54             if(currentSize >= fileSize){
55                 std::cout << "File receive success!" <<
filename << std::endl;
56                 file.close();
57             }
58         }
59         // send ACK to server
60         if(ackNum >= CUM_ACK) {
61             if(!randomLoss()){
62                 sendACK(recvBuffer.seq);
63             }
64             ackNum = 0;
65         }
66     }
67 }
68 // if package is not valid or receive seq is not equal to
current seq
69 // need to send last ACK again
70 else {
71     if(!randomLoss()) {
72         if(recvBuffer.isSYN()) {
73             sendACKSYN(EXPECT_SEQ - 1);
74         }
75         else if(recvBuffer.isFIN()) {
76             sendACKFIN(EXPECT_SEQ - 1);
77         }

```

```

78         else {
79             sendACK(EXPECT_SEQ - 1);
80         }
81     }
82 }
83 }
84 }
85 }

```

功能测试

由于在一开始代码设计的阶段并没有考虑用于测试的路由程序的影响，导致我在设计报文的时候使用了发送端和接收端的IP和端口号信息。

如果接入路由程序，发送端和接收端是分别连到路由程序的两个端口上的，也就是说其实双端之间是无感的。

这就导致我在计算校验和的时候，会因为发送端和接收端的端口号不一致导致问题。

因此在本次实验中并没有使用提供的路由程序，而是自己手动按照丢包率进行模拟丢包。模拟丢包的代码如下

```

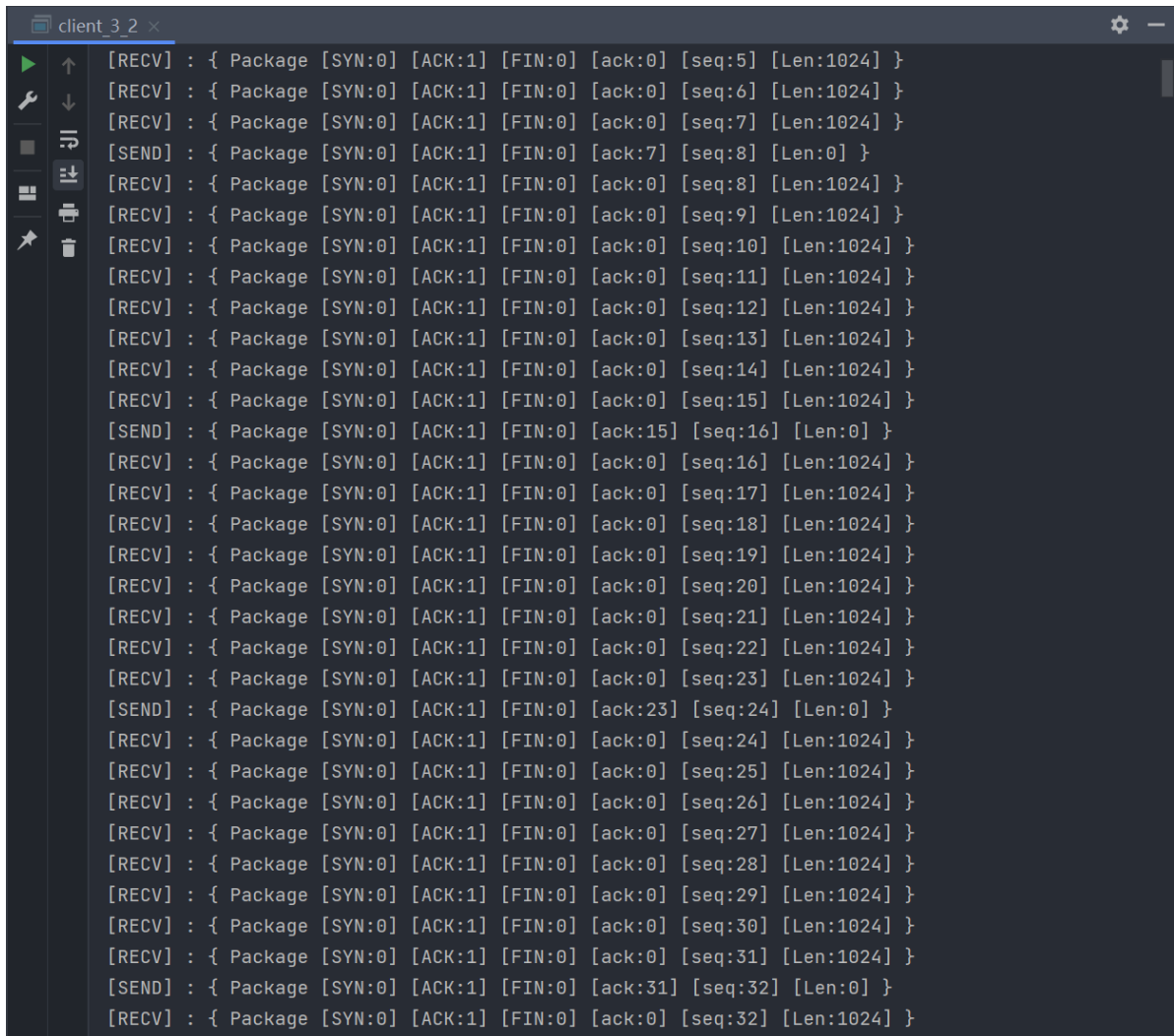
1 bool randomLoss() {
2     return rand() % 100 < LOSS_RATE;
3 }

```

接下来分别展示数据传输的效果和超时重传的效果。可以看到在下面的实验运行结果中清晰看到如下现象。

接收端的日志信息如图所示，从日志中我们可以看出，接收端并不是每次收到一个有效报文之后就会回应ACK。

而是当连续收到了8条有效报文之后，再统一确认这八条有效报文的最大序列号。



```
client_3.2 x
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:5] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:6] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:7] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:7] [seq:8] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:8] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:9] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:10] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:11] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:12] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:13] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:14] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:15] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:15] [seq:16] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:16] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:17] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:18] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:19] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:20] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:21] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:22] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:23] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:23] [seq:24] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:24] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:25] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:26] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:27] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:28] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:29] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:30] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:31] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:31] [seq:32] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:32] [Len:1024] }
```

发送端的日志信息如图所示，从日志中同样可以发现，发送端会流水线发送在发送缓冲区中的报文。

而当收到ACK之后，会根据确认号调整baseSeq的值。

当触发超时机制的时候，发送端会重发所有在发送缓冲区中的报文。

```
运行: server_3.2 x
[ACK] : Package (SEQ to : 3) sent successfully!
[TIMEOUT] : Package (SEQ from : 3 to : 18) re-send!
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:3] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:4] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:5] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:6] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:7] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:8] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:9] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:10] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:11] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:12] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:13] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:14] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:15] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:16] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:17] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:18] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:7] [seq:8] [Len:0] }
[ACK] : Package (SEQ to : 8) sent successfully!
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:19] [Len:1024] }
[Seg 18 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:20] [Len:1024] }
[Seg 19 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:21] [Len:1024] }
[Seg 20 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:22] [Len:1024] }
[Seg 21 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:23] [Len:1024] }
[Seg 22 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:15] [seq:16] [Len:0] }
[ACK] : Package (SEQ to : 16) sent successfully!
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:24] [Len:1024] }
[Seg 23 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:25] [Len:1024] }
```

最终展示成功收到的图像信息。

