

---

# 华中科技大学计算机学院

## 《计算机通信与网络》实验报告

班级 CS2007      姓名 孙溪      学号 U202015512

项目	Socket 编程 (40%)	数据可靠传输协议设计 (20%)	CPT 组网 (20%)	平时成绩 (20%)	总分
得分					

教师评语：

教师签名：

给分日期：

---

---

# 目 录

实验二 数据可靠传输协议设计实验.....	1
1.1 环境 .....	1
1.2 系统功能需求.....	1
1.3 系统设计 .....	2
1.4 系统实现.....	5
1.5 系统测试及结果说明 .....	14
1.6 其它需要说明的问题.....	18
1.7 参考文献.....	19
心得体会与建议 .....	20
2.1 心得体会 .....	20
2.2 建议 .....	20

---

## 实验二 数据可靠传输协议设计实验

### 1.1 环境

#### 1.1.1 开发平台

操作系统: Microsoft Windows10

处理器: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz

IDE: Microsoft Visual Studio Community 2019

第三方依赖: 模拟网络环境 netsimlib.lib

#### 1.1.2 运行平台

操作系统: Microsoft Windows10

处理器: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz

IDE: Microsoft Visual Studio Community 2019

第三方依赖: 模拟网络环境 netsimlib.lib

### 1.2 系统功能需求

可靠运输层协议只考虑单向传输, 即: 只有发送方发送数据报文, 接收方仅仅接收报文并给出确认报文。要求实现具体协议时, 指定编码报文序号的二进制位数 (例如 3 位二进制编码报文 序号) 以及窗口大小 (例如大小为 4), 报文段序号必须按照指定的二进制位数进行编码。代码实现不需要基于 Socket API, 不需要利用多线程, 不需要任何 UI 界面。

主要需要实现三个可靠运输层协议:

1. 实现基于 GBN 的可靠传输协议, 分值为 50%。
2. 实现基于 SR 的可靠传输协议, 分值为 30%。
3. 实现一个简化版的 TCP 协议, 分值 20%。这部分需要在 GBN 协议的基础上根据 TCP 的可靠数据传输机制, 添加支持快速重传和超时重传的功能。报文段序号需要按照报文段为单位进行编号, 确认号为收到的最后一个报文段序号。不需要考虑流量控制和拥塞控制。

## 1.3 系统设计

可靠传输协议整体设计：

设计发送方和接收方的报文段格式，每个报文段需要包含序号、校验和等信息。

实现发送方的发送报文段的过程，在发送报文段时，需要考虑窗口大小的限制，只有在窗口范围内的报文段才能被发送。发送方还需要开启一个计时器，作为重传依据。

实现接收方的接收报文段和发送确认报文的过程，接收方在接收到报文段后，需要验证校验和是否正确，如果正确则发送确认报文，并将报文段存储到缓冲区中。接收方也需要开启一个计时器，如果在计时器超时之前没有收到新的报文段，则接收方需要发送一个请求报文段的确认报文。

在发送方和接收方之间建立一个模拟的网络环境，利用模拟网络环境进行功能测试。

### 1.3.1 GBN

基于 GBN 的可靠传输协议的系统设计：

发送窗口大小为 N，接收窗口大小为 1。发送方按照指定的窗口大小连续发送帧，接收方进程跟踪它期望接收的下一帧的序列号，并在它发送的每个确认（ACK）中包含该编号。接收方将丢弃任何没有预期的确切序列号的帧（即已经确认的重复帧，或者它希望稍后接收的无序帧），并将重新发送最后一个正确的 ACK。GBNReceiver 和 GBNSender 设计如图 1、图 2 所示：

```
class GBNRdtReceiver :public RdtReceiver
{
private:
    int expectSequenceNumberRcvd;    // 期待收到的下一个报文序号
    Packet lastAckPkt;                // 上次发送的确认报文

public:
    GBNRdtReceiver();
    virtual ~GBNRdtReceiver();

public:
    void receive(const Packet& packet); // 接收报文，将被NetworkService调用
};
```

图 1 GBNReceiver 设计

```

class GBNRdtSender :public RdtSender
{
private:
    int base; //发送窗口的base
    int nextseqnum; // 下一个发送序号
    bool waitingState; // 是否处于等待Ack的状态
    Packet packetWaitingAck; //已发送并等待Ack的数据包
    Packet win[len];
    int num_packet_win;
public:
    bool getWaitingState();
    bool send(const Message& message); //发送应用层下来的Message
    //如果发送方成功地将Message发送到网络层，返回true;如果因为发送方处于等待正确确认状态而拒绝发送Message，则返回false
    void receive(const Packet& ackPkt); //接受确认Ack
    void timeoutHandler(int seqNum);

public:
    GBNRdtSender();
    virtual ~GBNRdtSender();
};

```

图 2 GBNSender 设计

### 1.3.2 SR

发送和接收窗口大小都大于 1，即使在帧丢失之后，发送过程也继续发送由窗口大小指定的多个帧。SRSender 和 SRReciever 设计如图 3、图 4 所示：

```

#define N 4//窗口大小
#define M 8//序号

class SrRdtSender :public RdtSender {
private:
    //序号采用3位编码，即0-7，采用模8操作
    int send_base; //最早未确认分组序号
    int nextseqnum; //下一个待发送分组序号
    Packet sndpkt[M];
    bool snd[M];
public:
    bool send(const Message& message); //发送应用层下一个的Message，由NetworkService调用，
    //如果发送方成功地将Message发送到网络层，返回true;
    //如果因为发送方处于等待确认状态或发送窗口已满而拒绝发送Message，则返回false
    void receive(const Packet& ackPkt); //接受确认Ack，将被NetworkService调用
    void timeoutHandler(int seqNum); //Timeout handler，将被NetworkService调用
    bool getWaitingState(); //返回RdtSender是否处于等待状态，如果发送方正等待确认或者发送窗口已满，返回true
public:
    SrRdtSender();
    ~SrRdtSender();
};

```

图 3 SRSender 设计

```

#define N 4//窗口大小
#define M 8//序号

class SrRdtReceiver :public RdtReceiver {
    bool rcv[M];//接受窗口是否收到
    Packet rcvpkt[M];//收到的packet
    int rcv_base;
    Packet ackpacket;//确认packet

public:
    void receive(const Packet& packet); //接收报文，将被NetworkService调用
    SrRdtReceiver();
    ~SrRdtReceiver();
};

```

图 4 SRReceiver 设计

### 1.3.3 TCP

结合了 GBN 和 SR 两种可靠传输协议，TCP 发送方仅需要维护已发送过但未被确认的最小序号和下一个要发送的字节的序号。但是，TCP 使用选择重传而不是窗口长度来限制流水线中未完成的、未被确认的分组。发送方只会重传未被确认的最小序号的分组，而不像 GBN 协议那样会重传最小序号之后的所有分组。TCPReceiver 和 TCPSender 如图 5、图 6 所示：

```

class TCPRdtReceiver :public RdtReceiver
{
private:
    int seq;
    Packet lastAckPkt;
public:
    TCPRdtReceiver();
    virtual ~TCPRdtReceiver();

public:
    void receive(const Packet& packet);
};

```

图 5 TCPReceiver 设计

```

#define len 8
class TCPRdtSender :public RdtSender
{
private:
    int base;
    int nextseqnum;    //下一个发送序号
    bool waitingState; //是否等待ack
    Packet win[len];   //窗口
    int num_pac_win;   //下一个存放在窗口中的序号
    int count; //快速重传判断
    int current_rcv_ack; //当前要收到的ack
    int last_rcv_ack;   //最后收到的ack
public:
    bool getWaitingState();
    bool send(const Message& message);
    void receive(const Packet& ackPkt);
    void timeoutHandler(int seqNum);

public:
    TCPRdtSender();
    virtual ~TCPRdtSender();
};

```

图 6 TCPSender 设计

## 1.4 系统实现

### 1.4.1 GBN

#### (1) GBNSender

发送端需要维护一个窗口大小，设置循环发送分组，判断 nextSeq 是否在窗口内，一次性发完一整个窗口内的分组并且等待对应序列号的 ACK 被返回。根据接收端返回的 ACK 编号调整窗口 base 的值为 ACK+1，如果达到了窗口最大值，关闭计时器，如果还不是，就设置定时器继续等待。如果超时，说明发送端没有在规定时间内收到预期 ACK，会重新发送这个序列号和其之后的所有分组。

(a) send 方法实现如图 7 所示：

```

bool GBNRdtSender::send(const Message& message) {
    if (nextseqnum < base + len) {
        this->waitingState = false; //标记此时窗口未滿
        this->win[num_packet_win].acknum = -1; //ack置为-1
        this->win[num_packet_win].seqnum = this->nextseqnum; //指向下一个报文
        this->win[num_packet_win].checksum = 0; //检查和置0
        memcpy(this->win[num_packet_win].payload, message.data, sizeof(message.data)); //拷贝数据

        this->win[num_packet_win].checksum = pUtils->calculateChecksum(this->win[num_packet_win]); //计算检查和
        pUtils->printPacket("发送方发送报文", this->win[num_packet_win]);
        if (base == nextseqnum) //若为窗口首元素则打开计时器
            pns->startTimer(SENDER, Configuration::TIME_OUT, this->win[num_packet_win].seqnum);

        pns->sendToNetworkLayer(RECEIVER, this->win[num_packet_win]); //发送报文至网络层
        this->num_packet_win++; //窗口内packet数目+1

        if (num_packet_win > len) //判断窗口是否滿
            this->waitingState = true; //窗口满了!

        this->nextseqnum++; //发送下一个组
        return true;
    }
    else {
        this->waitingState = true;
        return false;
    }
}

```

图 7 GBNSender::send 实现

(b) receive 方法实现如图 8 所示:



```

void GBNRdtSender::receive(const Packet& ackPkt) {
    if (this->num_packet_win > 0) { //窗口报文数大于0

        //检查校验和是否正确
        int checkSum = pUtils->calculateChecksum(ackPkt);

        //如果校验和正确，并且确认序号=发送方已发送并等待确认的数据包序号
        if (checkSum == ackPkt.checksum && ackPkt.acknum >= this->base) {

            int num = ackPkt.acknum - this->base + 1; //记录收到的ack序号

            base = ackPkt.acknum + 1;
            pUtils->printPacket("发送方正确收到确认", ackPkt);

            if (this->base == this->nextseqnum)
                pns->stopTimer(SENDER, this->win[0].seqnum); //结束则停止
            else { //重启计时器
                pns->stopTimer(SENDER, this->win[0].seqnum);
                pns->startTimer(SENDER, Configuration::TIME_OUT, this->win[num].seqnum);
            }

            for (int i = 0; i < num_packet_win - num; i++)
            {
                win[i] = win[i + num]; //将窗口内的packet向前移动num位
                printf("The current windows's %d number is %d\n", i, win[i].seqnum);
            }
            this->num_packet_win = this->num_packet_win - num; //窗口内包数目减去num
        }
    }
}

```

图 8 GBNSender::recieve 实现

## (2) GBNReceiver

只要是收到了预期的序列号分组，就留下这个分组并且预期自增 1，然后返回此序列号的 ACK；如果不是，直接丢弃并返回上一个序列号的 ACK。

(a) receive 方法如图 9 所示：

```

void GBNRdtReceiver::receive(const Packet& packet) {
    //检查校验和是否正确
    int checksum = pUtils->calculateChecksum(packet);

    //如果校验和正确，同时收到报文的序号等于接收方期待收到的报文序号一致
    if (checksum == packet.checksum && this->expectSequenceNumberRcvd == packet.seqnum) {
        pUtils->printPacket("接收方正确收到发送方的报文", packet);

        //取出Message，向上递交给应用层
        Message msg;
        memcpy(msg.data, packet.payload, sizeof(packet.payload));
        pns->deliverToAppLayer(RECEIVER, msg);

        lastAckPkt.acknum = packet.seqnum; //确认序号等于收到的报文序号
        lastAckPkt.checksum = pUtils->calculateChecksum(lastAckPkt); //计算ackpackage检查和

        pUtils->printPacket("接收方发送确认报文", lastAckPkt);
        pns->sendToNetworkLayer(SENDER, lastAckPkt); //调用模拟网络环境的sendToNetworkLayer，通过网络层发送确认报文到对方

        this->expectSequenceNumberRcvd++; //接收序号
    }
    else {
        if (this->expectSequenceNumberRcvd != packet.acknum) {
            pUtils->printPacket("接收方没有正确收到发送方的报文, 报文序号不对", packet);
        }
        else {
            pUtils->printPacket("接收方没有正确收到发送方的报文, 数据校验错误", packet);
        }
        pUtils->printPacket("接收方重新发送上次的确认报文", lastAckPkt);
        pns->sendToNetworkLayer(SENDER, lastAckPkt); //调用模拟网络环境的sendToNetworkLayer，通过网络层发送上次的确认报文
    }
}

```

图 9 GBNRdtReceiver::receive 实现

## 1.4.2 SR

发送和接收窗口的大小相等，并且是最大序列号的一半。

### (1) SRSENDER

发送方的事件与动作：

- 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送；否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。
- 超时。定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。
- 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。若该分组的序号等于 send\_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

(a) send 方法实现如图 10 所示：

```

bool SrRdtSender::send(const Message& message) {
    if (getWaitingState() == true)
        return false;
    snd[nextseqnum] = false; //未确认
    sndpkt[nextseqnum].acknum = -1; //忽略该字段

    sndpkt[nextseqnum].seqnum = nextseqnum; //当前包的序号即为nextseqnum
    sndpkt[nextseqnum].checksum = 0; //校验和为0
    memcpy(sndpkt[nextseqnum].payload, message.data, sizeof(message.data));
    //message打包
    sndpkt[nextseqnum].checksum = pUtils->calculateCheckSum(sndpkt[nextseqnum]);
    //计算当前包的校验和
    pUtils->printPacket("发送方发送报文", sndpkt[nextseqnum]);
    pns->stopTimer(SENDER, nextseqnum);
    pns->startTimer(SENDER, Configuration::TIME_OUT, nextseqnum); //为第一个元素设置时钟
    pns->sendToNetworkLayer(RECEIVER, sndpkt[nextseqnum]); //发送数据
    nextseqnum = (nextseqnum + 1) % M; //后移

    return true;
}

```

图 10 SRSender::send 实现

(b) receive 方法实现如图 11 所示：

```

void SrRdtSender::receive(const Packet& ackPkt) {
    if (send_base != nextseqnum) { //base没确认完
        //检查校验和是否正确
        int checksum = pUtils->calculateChecksum(ackPkt);
        if (checksum == ackPkt.checksum) {
            if (((send_base + N - 1) % M > send_base && (ackPkt.acknum >= send_base && ackPkt.acknum <= (send_base + N - 1) % M))
                || ((send_base + N - 1) % M < send_base && ((ackPkt.acknum >= send_base && ackPkt.acknum <= M - 1)
                || ackPkt.acknum <= (send_base + N - 1) % M)))
            {
                snd[ackPkt.acknum] = true; //标记为已确认
                pns->stopTimer(SENDER, ackPkt.acknum);
                pUtils->printPacket("发送方正确收到确认", ackPkt);
                if (send_base == ackPkt.acknum) {
                    int i = send_base;
                    for (; i != nextseqnum; )
                    {
                        if (snd[i] == false) break;
                        i = (i + 1) % M;
                    }
                    cout << "窗口移动前: " << endl;
                    for (int i = send_base; i != nextseqnum; ) {
                        //输出窗口
                        Message msg;

                        memcpy(msg.data, sndpkt[i].payload, sizeof(sndpkt[i].payload));
                        msg.data[21] = '\0';
                        cout << msg.data << endl;
                        i = (i + 1) % M;
                    }
                    cout << endl;
                    send_base = i;
                    cout << "窗口移动后: " << endl;
                    for (int i = send_base; i != nextseqnum; ) {
                        //输出窗口
                        Message msg;
                        memcpy(msg.data, sndpkt[i].payload, sizeof(sndpkt[i].payload));
                        msg.data[21] = '\0';
                        cout << msg.data << endl;
                        i = (i + 1) % M;
                    }
                    cout << endl;
                }
            }
        }
    }
}

```

图 11 SRSender::receive 实现

## (2) SRReceiver

接收方的事件与动作:

- 序号在  $[rcv\_base, rcv\_base+N-1]$  内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收端的基序号 ( $rcv\_base$ )，则该分组以及以前缓存的序号连续的 (起始于  $rcv\_base$  的) 分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。

- 序号在  $[rcv\_base-N, rcv\_base-1]$  内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前确认过的分组。

- 其他情况。忽略该分组。

(a) receive 方法如图 12 所示:

```

void SrRdtReceiver::receive(const Packet& packet) {
    //检查校验和是否正确
    int checksum = pUtils->calculateChecksum(packet);
    //如果校验和正确, 同时收到报文的序号等于接收方期待收到的报文序号一致
    if (checksum == packet.checksum) {
        pUtils->printPacket("接收方正正确收到发送方的报文", packet);

        if (((rcv_base + N - 1) % M > rcv_base && (packet.seqnum >= rcv_base && packet.seqnum <= (rcv_base + N - 1) % M))
            || ((rcv_base + N - 1) % M < rcv_base && ((packet.seqnum >= rcv_base && packet.seqnum <= M - 1) || packet.seqnum <= (rcv_base + N - 1) % M)))
        {
            ackpacket.acknum = packet.seqnum; //确认序号等于收到的报文序号
            ackpacket.checksum = pUtils->calculateChecksum(ackpacket);
            pUtils->printPacket("接收方发送确认报文", ackpacket);
            pns->sendToNetworkLayer(SENDER, ackpacket); //调用模拟网络环境的sendToNetworkLayer, 通过网络层发送确认报文到对方
            if (rcv[packet.seqnum] == false) {
                rcvpkt[packet.seqnum] = packet; //存入
                rcv[packet.seqnum] = true; //已收到
            }
            if (packet.seqnum == rcv_base) {
                int i = rcv_base;
                for (; i != (rcv_base + N) % M; ) {
                    if (rcv[i] == true) //已收到
                    {
                        //取出Message, 向上递交给应用层
                        Message msg;
                        memcpy(msg.data, rcvpkt[i].payload, sizeof(rcvpkt[i].payload));
                        pns->deliverToAppLayer(RECEIVER, msg);
                        rcv[i] = false;
                        i = (i + 1) % M;
                    }
                    else {
                        break;
                    }
                }
                cout << "接收窗口移动前: " << endl;
                for (int j = rcv_base; j != (rcv_base + N) % M; ) {
                    Message msg;
                    memcpy(msg.data, rcvpkt[j].payload, sizeof(rcvpkt[j].payload));
                    msg.data[21] = '\0';
                    cout << msg.data << endl;

                    j = (j + 1) % M;
                }
                cout << endl;
                rcv_base = i;
                cout << "接收窗口移动后: " << endl;
                for (int j = rcv_base; j != (rcv_base + N) % M; ) {
                    //输出窗口
                    Message msg;
                    memcpy(msg.data, rcvpkt[j].payload, sizeof(rcvpkt[j].payload));
                    msg.data[21] = '\0';
                    cout << msg.data << endl;

                    j = (j + 1) % M;
                }
                cout << endl;
            }
        }
    }
    else if (((rcv_base - N + 8) % M < (rcv_base - 1 + 8) % M && (packet.seqnum >= (rcv_base - N + 8) % M && packet.seqnum <= (rcv_base - 1 + 8) % M))
        || ((rcv_base - N + 8) % M > (rcv_base - 1 + 8) % M && ((packet.seqnum >= (rcv_base - N + 8) % M && packet.seqnum <= M - 1) || packet.seqnum <= (rcv_base - 1 + 8) % M)))
    {
        ackpacket.acknum = packet.seqnum; //确认序号等于收到的报文序号
        ackpacket.checksum = pUtils->calculateChecksum(ackpacket);
        pUtils->printPacket("接收方发送确认报文", ackpacket);
        pns->sendToNetworkLayer(SENDER, ackpacket); //调用模拟网络环境的sendToNetworkLayer, 通过网络层发送确认报文到对方
    }
}

```

图 12 SRReceiver::receive 实现

### 1.4.3 TCP

#### (1) TCPSender

发送方的事件与动作:

使用发送窗口来控制发送报文段的数量, 在收到确认报文后, 通过更新窗口的起始序号和结束序号来移动窗口, 并发送新的报文段。使用超时重传机制来处理报文段发送失败的情况。当超时时间到达时, 如果还没有收到确认报文, 则重新发送报文段。使用单一的超时计时器来处理超时重传, 不需要估算 RTT 动态调整定时器 Timeout 参数。

(a) send 方法实现如图 13 所示:

```

}bool TCPRdtSender::send(const struct Message& message)
{
    if (nextseqnum < base + len)          //窗口未满
    {
        this->waitingState = false;
        this->win[num_pac_win].acknum = -1;
        this->win[num_pac_win].seqnum = this->nextseqnum;
        this->win[num_pac_win].checksum = 0;
        memcpy(this->win[num_pac_win].payload, message.data, sizeof(message.data)); //获取应用层数据
        this->win[num_pac_win].checksum = pUtils->calculateChecksum(this->win[num_pac_win]); //计算检查和

        pUtils->printPacket("发送方发送报文", this->win[num_pac_win]);
        if (base == nextseqnum) //base的报文, 则启动定时器
            pns->startTimer(SENDER, Configuration::TIME_OUT, this->win[num_pac_win].seqnum);
        pns->sendToNetworkLayer(RECEIVER, this->win[num_pac_win]);

        this->num_pac_win++; //窗口内packet+1

        if (num_pac_win > len) //满了
            this->waitingState = true;

        this->nextseqnum++;
        return true;
    }

    else //窗口满
    {
        this->waitingState = true;
        return false;
    }
}

```

图 13 TCPSender::send 实现

(b) receive 方法如图 14 所示:

```

void TCPRdtSender::receive(const struct Packet& ackPkt)
{
    if (this->num_pac_win > 0) //窗口内有待确认的报文
    {
        int checkSum = pUtils->calculateChecksum(ackPkt); //计算检查和

        printf("receive_ACK number: %d\n", ackPkt.acknum);
        printf("base number: %d\n", this->base);

        if (checkSum == ackPkt.checksum && ackPkt.acknum >= this->base) //ack_num比base更大
        {
            if (ackPkt.acknum == this->win[0].seqnum) //判断现在的序号和上一次的序号是否相同
            {
                this->count++; //计数+1
                if (count == 4) //如果ack四次，则重传当前窗口的第一个报文
                {
                    pns->stopTimer(SENDER, this->win[0].seqnum);
                    pns->sendToNetworkLayer(RECEIVER, this->win[0]); //将第一个报文发送给接收方
                    pUtils->printPacket("\n冗余ACK*4，快速重传当前窗口第一个报文", win[0]);
                    pns->startTimer(SENDER, Configuration::TIME_OUT, this->win[0].seqnum);
                    printf("\n冗余ACK%d *4 \n", ackPkt.acknum);
                    this->count = 0;
                    return;
                } //冗余ack快速重传
            }

            else {
                this->count = 1;
            }

            if (this->count != 1)
                return;

            else
            {
                int num = ackPkt.acknum - this->base;
                base = ackPkt.acknum; //发送方的ack为期待收到的base报文的序号
                pUtils->printPacket("接收到ACK报文", ackPkt);

                if (this->base == this->nextseqnum) //如果收到的确认是发送窗口中的最后一个报文的确认
                    pns->stopTimer(SENDER, this->win[0].seqnum);
                else
                {
                    pns->stopTimer(SENDER, this->win[0].seqnum);
                    pns->startTimer(SENDER, Configuration::TIME_OUT, this->win[num].seqnum); //启动新的计时器
                }
                for (int i = 0; i < num_pac_win - num; i++)
                {
                    win[i] = win[i + num];
                    printf("The current windows' s %d number is %d\n", i, win[i].seqnum);
                }
                this->num_pac_win = this->num_pac_win - num; //平移窗口
            }
        }
    }
}

```

图 14 TCPSender::receive 实现

## (2) TCPReciever

接收方的事件与动作：

在收到方，使用接收窗口来控制接收报文段的数量，并使用确认报文来告诉发送方哪些报



文段已经收到。使用快速重传机制来处理报文段丢失的情况。当收到重复的报文段时，发送确认报文确认之前的所有报文段。使用确认号来确定最后一个收到的报文段的序号。

(a) receive 方法实现如图 15 所示：

```
void TCPRdtReceiver::receive(const struct Packet& packet)//累计确认
{
    int checksum = pUtils->calculateChecksum(packet);//检查发送方发送的报文是否出错
    if (checksum == packet.checksum && this->seq == packet.seqnum) //如果正确并且为期待接受的报文
    {
        pUtils->printPacket("接收方正确收到报文", packet);
        Message msg;
        memcpy(msg.data, packet.payload, sizeof(packet.payload));
        pns->deliverToAppLayer(RECEIVER, msg);//向上递交给应用层

        this->lastAckPkt.acknum = this->seq + 1;//发送的ack更新期待的序列号
        this->lastAckPkt.checksum = pUtils->calculateChecksum(lastAckPkt);//重新计算校验和
        pUtils->printPacket("接收方向发送方发送确认报文", lastAckPkt);
        pns->sendToNetworkLayer(SENDER, lastAckPkt);//发送ack至网络层
        this->seq++; //期待的下一个报文序号+1
    }

    else//如果失序或者出错
    {
        if (packet.acknum != seq)
            pUtils->printPacket("ERROR: 接收方未正确收到报文: 报文序列号错误", packet);

        else
            pUtils->printPacket("ERROR: 接收方未正确收到报文: 校验和错误", packet);

        pUtils->printPacket("ERROR: 接收方发送冗余ACK", this->lastAckPkt);//lastpacket中的acknum为seq的值，即期待接受到
        pns->sendToNetworkLayer(SENDER, lastAckPkt);
    }
}
```

图 15 TCPReceiver::receive 实现

## 1.5 系统测试及结果说明

### 1.5.1 GBN

经多次测试，此协议能够达到预期效果。每次移动滑动窗口时在控制台输出滑动窗口的内容滑动窗口的移动正确。控制台输出如图 16 所示，脚本测试结果如图 17 所示。



```
Microsoft Visual Studio 调试控制台
The current windows's 0 number is 104
The current windows's 1 number is 105
接收方没有正确收到发送方的报文, 报文序号不对: seqnum = 103, acknum = -1, checksum = 33308, YYYYYYYYYYYYYYYYYYYY
接收方重新发送上次的确认报文: seqnum = -1, acknum = 103, checksum = 12748, .....
接收方正确收到发送方的报文: seqnum = 104, acknum = -1, checksum = 30737, ZZZZZZZZZZZZZZZZZZZZZ
*****模拟网络环境*****: 向上递交给应用层数据: ZZZZZZZZZZZZZZZZZZZZZ
接收方发送确认报文: seqnum = -1, acknum = 104, checksum = 12747, .....
接收方正确收到发送方的报文: seqnum = 105, acknum = -1, checksum = 29768, EOF
*****模拟网络环境*****: 向上递交给应用层数据: EOF
接收方发送确认报文: seqnum = -1, acknum = 105, checksum = 12746, .....
发送方正确收到确认: seqnum = -1, acknum = 104, checksum = 12747, .....
The current windows's 0 number is 105
发送方定时器时间到, 重发上次发送的报文: seqnum = 105, acknum = -1, checksum = 29768, EOF
接收方没有正确收到发送方的报文, 报文序号不对: seqnum = 105, acknum = -1, checksum = 29768, EOF
接收方重新发送上次的确认报文: seqnum = -1, acknum = 105, checksum = 12746, .....
发送方正确收到确认: seqnum = -1, acknum = 105, checksum = 12746, .....
*****模拟网络环境*****: 模拟网络环境已发送完应用层数据, 关闭模拟网络环境
已发送应用层Message个数: 105
发送到网络层数据Packet个数: 239
网络层丢失的数据Packet个数: 23
网络层损坏的数据Packet个数: 29
发送到网络层确认Packet个数: 216
网络层丢失的确认Packet个数: 18
网络层损坏的确认Packet个数: 21

C:\Users\sucy\Desktop\lab\lab2\GBN\Debug\GBN.exe (进程 15556) 已退出, 代码为 0。
要在调试停止时自动关闭调试控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 16 GBN 控制台输出

输入文件和输出文件的内容每次相同:

```
C:\WINDOWS\system32\cmd.exe
Test "GBN.exe" 1:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 2:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 3:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 4:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 5:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 6:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 7:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 8:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 9:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "GBN.exe" 10:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

请按任意键继续. . .
```

图 17 GBN 输出结果测试

### 1.5.2 SR

经多次测试，此协议能够达到预期效果。每次移动滑动窗口时在控制台输出滑动窗口的内容，滑动窗口的内容正确，同时将控制台的输出重定向到文件 `result.txt` 中。控制台输出如图 18 所示，脚本测试结果如图 19 所示。

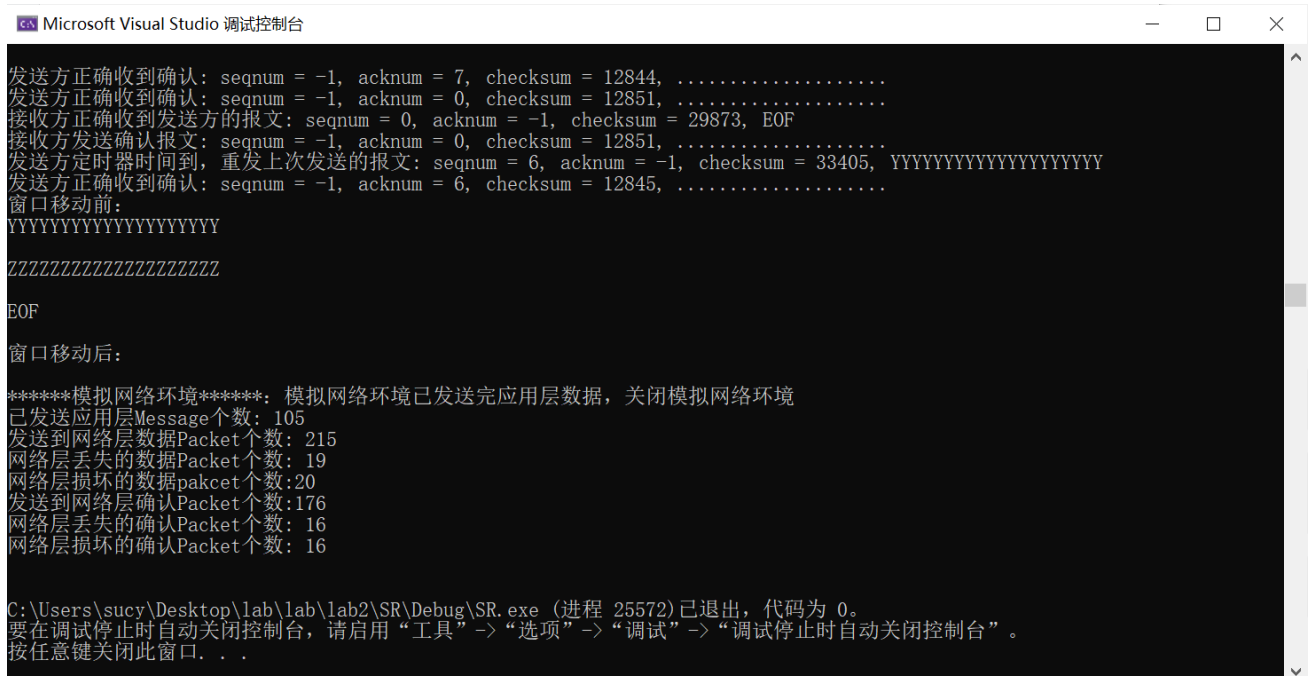


图 18 SR 控制台输出

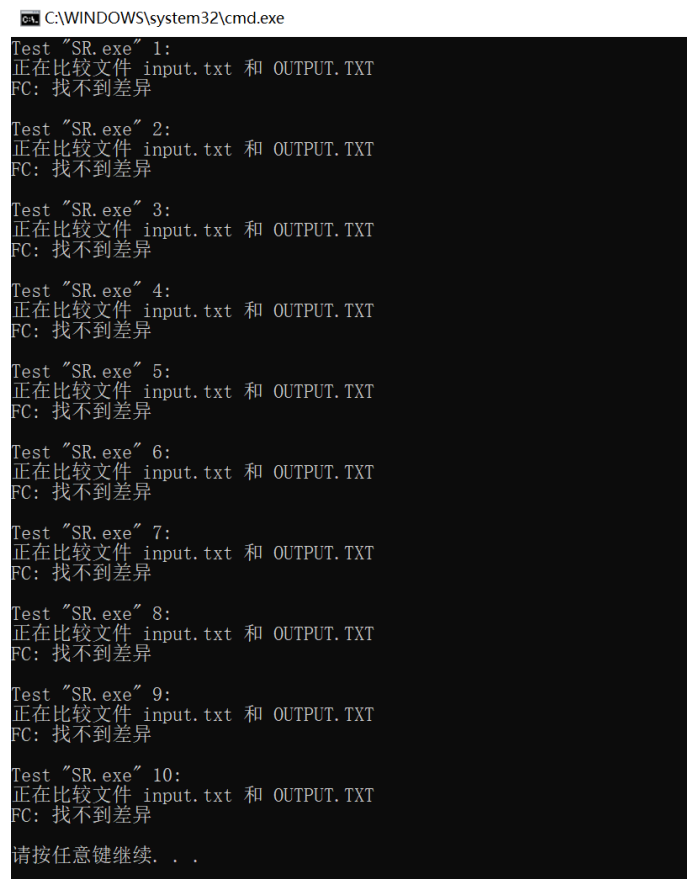
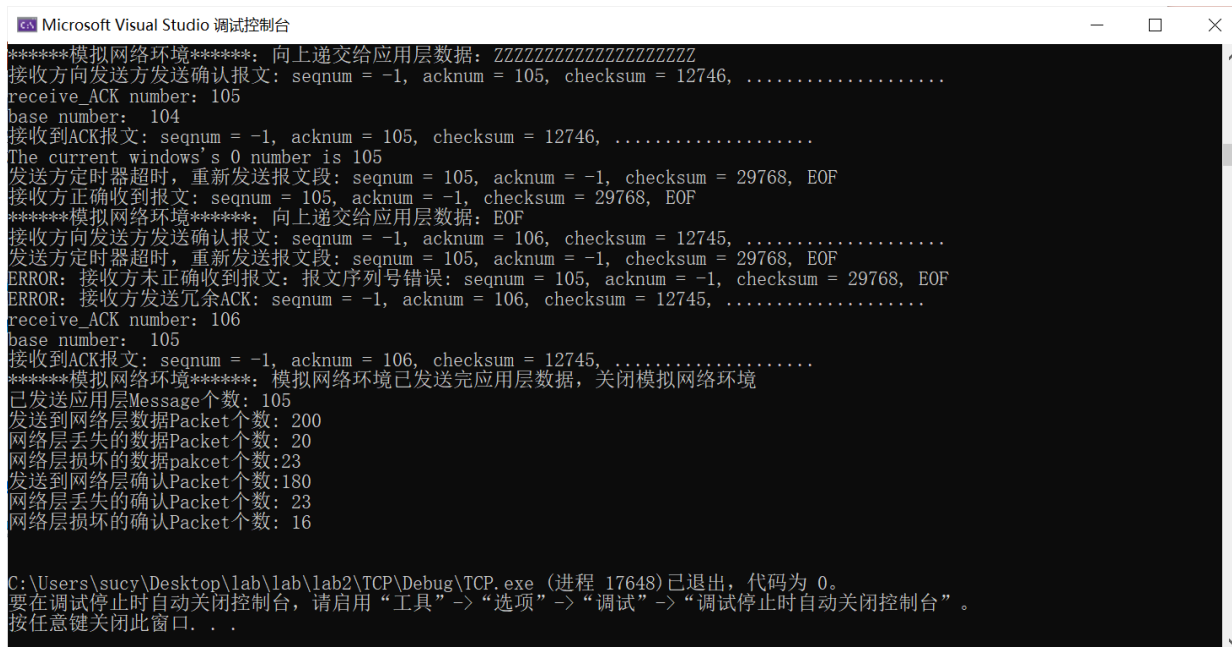


图 19 SR 输出结果测试

### 1.5.3 TCP

经多次测试，此协议能够达到预期效果。每次移动滑动窗口时在控制台输出滑动窗口的内容，滑动窗口的内容正确，同时将控制台的输出重定向到文件 `result.txt` 中，每次快速重传时在控制台输出提示信息。控制台输出如图 20 所示，脚本测试结果如图 21 所示。



```
Microsoft Visual Studio 调试控制台
*****模拟网络环境*****: 向上递交给应用层数据: ZZZZZZZZZZZZZZZZZZZ
接收方向发送方发送确认报文: seqnum = -1, acknum = 105, checksum = 12746, .....
receive_ACK number: 105
base number: 104
接收到ACK报文: seqnum = -1, acknum = 105, checksum = 12746, .....
The current windows' s 0 number is 105
发送方定时器超时, 重新发送报文段: seqnum = 105, acknum = -1, checksum = 29768, EOF
接收方正确收到报文: seqnum = 105, acknum = -1, checksum = 29768, EOF
*****模拟网络环境*****: 向上递交给应用层数据: EOF
接收方向发送方发送确认报文: seqnum = -1, acknum = 106, checksum = 12745, .....
发送方定时器超时, 重新发送报文段: seqnum = 105, acknum = -1, checksum = 29768, EOF
ERROR: 接收方未正确收到报文: 报文序列号错误: seqnum = 105, acknum = -1, checksum = 29768, EOF
ERROR: 接收方发送冗余ACK: seqnum = -1, acknum = 106, checksum = 12745, .....
receive_ACK number: 106
base number: 105
接收到ACK报文: seqnum = -1, acknum = 106, checksum = 12745, .....
*****模拟网络环境*****: 模拟网络环境已发送完应用层数据, 关闭模拟网络环境
已发送应用层Message个数: 105
发送到网络层数据Packet个数: 200
网络层丢失的数据Packet个数: 20
网络层损坏的数据packet个数: 23
发送到网络层确认Packet个数: 180
网络层丢失的确认Packet个数: 23
网络层损坏的确认Packet个数: 16

C:\Users\sucy\Desktop\lab\lab\lab2\TCP\Debug\TCP.exe (进程 17648) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 20 TCP 控制台输出

```
C:\WINDOWS\system32\cmd.exe
Test "TCP.exe" 1:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 2:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 3:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 4:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 5:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 6:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 7:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 8:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 9:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

Test "TCP.exe" 10:
正在比较文件 input.txt 和 OUTPUT.TXT
FC: 找不到差异

请按任意键继续. . .
```

图 21 TCP 输出结果测试

## 1.6 其它需要说明的问题

3 种协议的测试脚本：check\_win.bat

set appname="TCP.exe" ::不同测试协议的执行文件名有所不同

set inputname="input.txt"

set outputname="output.txt"

set resultname="result.txt"

for /l %%i in (1,1,10) do (

    echo Test %appname% %%i:

    %appname% > %resultname% 2>&1

    fc /N %inputname% %outputname%

)

pause

---

## 1.7 参考文献

无

---

## 心得体会与建议

### 2.1 心得体会

在进行 Socket 编程实验时了解了使用 Socket 进行通信以及如何使用 Socket 编写网络程序。学习了解不同的 Socket 类型、如何创建 Socket 实例、如何绑定 Socket 到特定的端口以及如何使用 Socket 进行通信。并进行了异常的捕捉和处理，例如请求 404；完成符合要求的 Web 服务器的开发工作，能够完成可靠数据传输协议的实现工作

在进行可靠数据传输协议设计实验时，通过模拟网络环境来设计和测试可靠数据传输协议。使用一组标准的测试数据来模拟真实的网络环境，并使用不同的参数来测试协议的性能。协议需要考虑到网络中可能存在的丢包、乱序等问题，并采取相应的措施来确保数据能够正确地网络传输。在 StopWait 协议的基础上设计 GBN，对比不同协议的特性设计 SR，结合 GBN 和 SR 实现快速重传的 TCP，循序渐进实现更为复杂的协议需求，实现相应功能并进行功能测试。

在基于 CPT 的组网实验中，学会了如何使用 CPT 来设计网络协议，并了解了网络协议的工作原理。了解了网络仿真工具 Cisco Packet Tracer 的使用原理和方法，完成局域网和广域网的仿真组建工作，对仿真设备进行配置，使得仿真网络能够达到预期的实验目标。使用 CPT 来构建复杂的网络拓扑结构，并使用不同的协议进行数据传输。通过 CPT，我们可以更好地理解计算机网络的工作原理，并且可以方便地测试不同的网络设计方案。

### 2.2 建议

收获很多，无特别的建议。谢谢课程组老师的实验设计。