《计算机通信与网络》

实验指导手册

（可靠数据传输协议设计实验分册）

华中科技大学计算机科学与技术学院

二零二零年十月

目录

第一章实验目标和内容

1.1 实验目的

通过该实验了解和掌握运输层可靠数据传输原理以及具体实现方法。

1.2 实验环境

- 语言工具：基于 C++语言实现。
- 操作系统：Windows、Linux 操作系统。
- 其它要求：基于模拟网络环境 API 实现。

1.3 实验要求

- 可靠运输层协议实验只考虑单向传输，即：只有发送方发生数据报文，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。
- 提交实验设计报告和源代码；实验设计报告必须按照实验报告模板完成，源代码必须加详细注释。
- 代码编译运行成功后，运行给实验指导老师或者助教检查。

1.4 实验内容

本实验包括三个级别的内容，具体包括：

- 实现基于 GBN 的可靠传输协议，分值为 50%。
- 实现基于 SR 的可靠传输协议，分值为 30%。
- 在实现 GBN 协议的基础上，根据 TCP 的可靠数据传输机制实现一个简化版的 TCP 协议，分值 20%，要求：
  - ✓ 报文段格式、接收方缓冲区大小和 GBN 协议一样保持不变；
  - ✓ 报文段序号按照报文段为单位进行编号；
  - ✓ 单一的超时计时器，不需要估算 RTT 动态调整定时器 Timeout 参数；
  - ✓ 支持快速重传和超时重传，重传时只重传最早发送且没被确认的报文段；
  - ✓ 确认号为收到的最后一个报文段序号；
  - ✓ 不考虑流量控制、拥塞控制。

第二章模拟网络环境介绍

由于需要模拟真实网络环境下实际可能发生的丢包、报文损坏的情况，因此开发了一个模拟的网络环境。模拟网络环境模拟实现了应用层和网络层，而需要和学生实现的运输层 Rdt 协议协同工作完成数据的可靠传输。

2.1 模拟网络环境功能

模拟的网络环境实现了以下功能：

（1）应用层的数据向下递交给发送方运输层 Rdt 协议；

（2）接收方运输层 Rdt 协议收到差错检测无误的报文后向上层应用层递交；

（3）将发送方运输层 Rdt 协议准备好的报文通过网络层递交给接收方，在递交过程中按一定的概率会产生丢包、报文损坏；

（4）定时器的启动、关闭、定时器 Timeout 后通知发送方运输层 Rdt 协议。

2.2 模拟网络环境架构

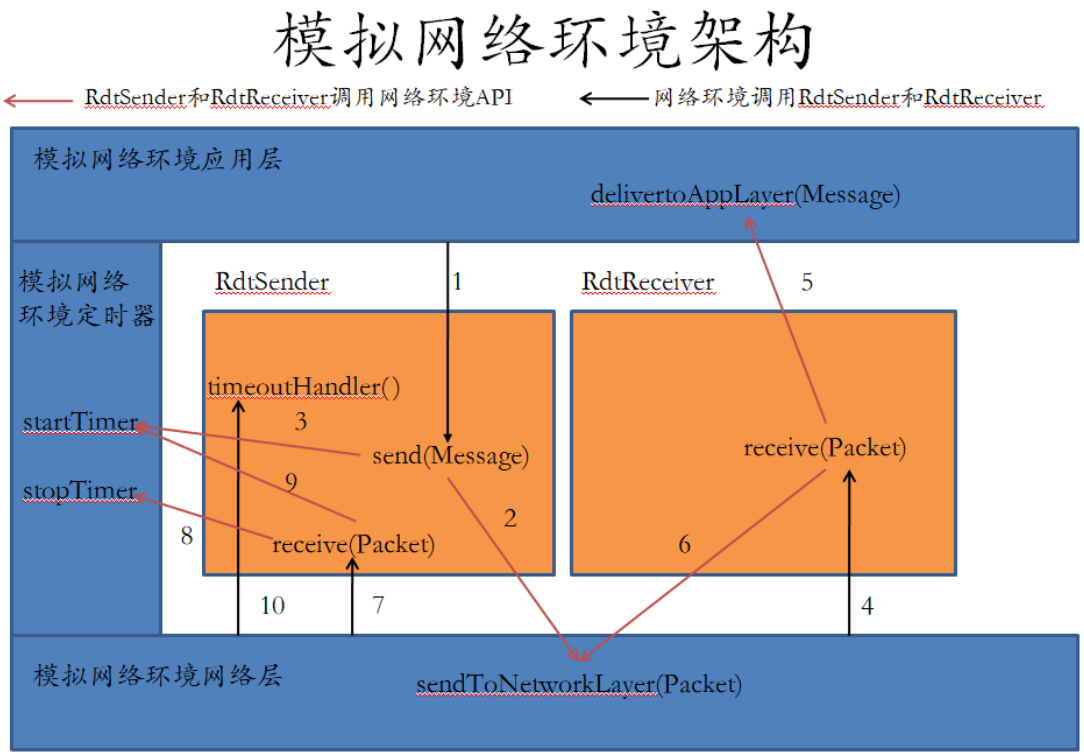

图 1 模拟网络环境架构

图 1 给出了模拟网络环境架构和学生实现的 Rdt 协议的之间的关系。二者之间需要协同工作。蓝色背景部分为模拟网络环境，橙色背景部分为 Rdt 协议的发送方（RdtSender）和

接收方（RdtReceiver）。红色箭头表示 RdtSender 和 RdtReceiver 调用模拟网络环境的函数，黑色箭头表示模拟环境调用 RdtSender 和 RdtReceiver 的函数。

模拟网络环境实现了以下函数以供 Rdt 协议调用：delivertoAppLayer(Message) 、sendToNetworkLayer(Packet) 、startTimer(int seqNum)、stopTimer(int seqNum)，这些函数功能将在 2.3 节介绍。

Rdt 协议的发送方 RdtSender 和 Rdt 协议的接收方则是学生需要实现的功能，它们共同实现了 Rdt 协议。其中 RdtSender 必须实现三个函数：send(Message)、receive(Packet)、timeoutHandler(int seqNum)；RdtReceiver 则必须实现一个函数：receive(Packet)，这些函数的功能将在 2.3 节介绍。

2.3 模拟网络环境与学生实现的代码之间的调用关系

模拟网络环境模拟实现了应用层和网络层，而需要和学生实现的运输层 Rdt 协议协同工作完成数据的可靠传输。从应用层有数据到来开始，它们之间的调用关系如下所述：

（1）模拟网络环境模拟产生应用层数据，调用 Rtdsender 的 Send(Message)方法；

（2）RtdSender 的 Send(Message)方法调用模拟网络环境的 sendToNetworkLayer(Packet)方法，将数据发送到模拟网络环境的网络层；

（3）RtdSender 的 Send(Message)方法调用模拟网络环境的 startTimer( )方法启动定时器；

（4）模拟网络环境调用 RdtReceiver 的 receive(Packet)方法将数据交给 RdtReceiver；

（5）如果校验正确，RdtReceiver 调用模拟网络环境的 delivertoAppLayer(Message)方法将数据向上递交给应用层；

（6）RdtReceiver 调用模拟网络环境的 sendToNetworkLayer(Packet)方法发送确认；

（7）模拟网络环境调用 RtdSender 的 receive(Packet)方法递交确认给 RtdSender；

（8）如果确认正确，RtdSender 调用模拟网络环境的 stopTimer 方法关闭定时器；

（9）如果确认不正确，RtdSender 调用模拟网络环境的 startTimer 方法重启定时器；

（10）如果定时器超时，模拟网络环境调用 RtdSender 的 timeoutHandler( )方法。

第三章数据结构、接口定义与模拟网络环境 API 介绍

3.1 数据结构定义

DataStructure.h 头文件里定义了应用层消息 Message 和运输层 Packet 类，同时定义了基本的参数，具体如下：

```
struct Configuration{

    /**
    定义各层协议Payload数据的大小（字节为单位）
    */
    static constint PAYLOAD_SIZE = 21;

    /**
    定时器时间
    */
    static constint TIME_OUT =20;

};

/**
    第五层应用层的消息
*/
struct Message {
    chardata[Configuration::PAYLOAD_SIZE];          //payload

    Message();
    Message(const Message &msg);
    Message& operator=(const Message &msg);
    ~Message();

    void print();
};

/**
    第四层运输层报文段
*/
struct Packet {
    int seqnum;                                     //序号
    int acknum;                                     //确认号
    int checksum;                                   //校验和
    char payload[Configuration::PAYLOAD_SIZE];      //payload
```

```
    Packet();
    Packet(const Packet &pkt);
    Packet &operator=(const Packet &pkt);
    bool operator==(const Packet &pkt) const;
    ~Packet();

    void print();
};
```

3.2 RdtSender 和 RdtReceiver 接口定义

在头文件 RdtSender.h 和 RdtReceiver.h 里分别定义 Rdt 协议发送方和接收方必须实现的接口，是通过抽象类 RdtSender 和 RdtReceiver 进行接口的定义，具体分别如下：

```
//定义RdtSender抽象类，规定了必须实现的三个接口方法
//具体的子类比如StopWaitRdtSender、GBNRdtSender必须给出这三个方法的具体实现
//只考虑单向传输，即发送方只发送数据和接受确认
struct RdtSender
{
    //发送应用层下来的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
    virtual bool send(Message &message) = 0;

    //接受确认Ack，将被NetworkService调用
    virtual void receive(Packet &ackPkt) = 0;

    //Timeout handler，将被NetworkService调用
    virtual void timeoutHandler(int seqNum) = 0;

    //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
    virtual bool getWaitingState() = 0;
};
```

这里需要特别说明的是：

（1）timeoutHandler 方法的参数 seqNum 为和该定时器关联的 Packet 的序号。虽然对于 StopWait、GBN、简化版的 TCP 协议只需要一个定时器，但是该函数还是需要一个序号参数，只不过该序号是最早发出但没有被确认的 Packet 的序号。但是如果实现 SR 协议，那么该参数就有具体的意义了：它指明了是哪个 Packet 的定时器超时了。

（2）getWaitingState 函数返回 RdtSender 是否处于等待状态，对于具体的 Rdt 协议，是否处于等待状态具有不同的含义：例如对于 StopWait 协议，当发送方等待上层发送的 Packet 的确认时，getWaitingState 函数应该返回 true；对于 GBN 协议，当发送方的发送窗口满了时，

getWaitingState 函数应该返回 true。定义这个接口方法的原因是模拟网络环境需要调用 RdtSender 的这个方法来判断是否需要将应用层下来的数据递交给 Rdt，这样学生实现 RdtSender 时不需要在内部维护一个 Packet 队列了，因为当 getWaitingState 返回 true 时，应用层不会有数据下来。RdtReceiver 的定义则为：

```
//定义RdtReceiver抽象类，规定了必须实现的一个接口方法
//具体的子类比如StopWaitRdtReceiver、GBNRdtReceiver必须给出这一个方法的具体实现
//只考虑单向传输，即接收方只接收数据
struct RdtReceiver
{
    virtual void receive(Packet &packet) = 0;          //接收报文，将被NetworkService调用
};
```

RdtReceiver 的接口定义就简单的多，这里不再解释。具体的 Rdt 协议实现类必须继承这二个抽象类，这样就保证了不同的 Rdt 协议接口一致性。否则具体 Rdt 协议实现类的对象无法注入到模拟网络环境一起协同工作。

3.3 模拟网络环境 API 接口定义

模拟网络环境 API 接口定义了学生实现的具体 Rdt 协议实现类可以调用的函数，具体定义在 NetworkService.h 头文件中定义：

```
/定义NetworkService抽象类，规定了学生实现的RdtSender和RdtReceiver可以调用的的接口方法
struct NetworkService {
    //发送方启动定时器，由RdtSender调用
    virtual void startTimer(RandomEventTarget target, int timeOut,int seqNum) = 0;

    //发送方停止定时器，由RdtSender调用
    virtual void stopTimer(RandomEventTarget target,int seqNum) = 0;

    //将数据包发送到网络层，由RdtSender或RdtReceiver调用
    virtual void sendToNetworkLayer(RandomEventTarget target, Packet pkt) = 0;

    //将数据包向上递交到应用层，由RdtReceiver调用
    virtual void delivertoAppLayer(RandomEventTarget target, Message msg) = 0;

    //初始化网络环境，在main里调用
    virtual void init() = 0;
    //启动网络环境，在main里调用
    virtual void start() = 0;
    //注入具体的发送方对象，在main里调用
    virtual void setRtdSender(RdtSender *ps) = 0;

    //设置具体的接收对象，在main里调用
    virtual void setRtdReceiver(RdtReceiver *ps) = 0;
```

```
    //设置输入文件路径
    virtual void setInputFile(const char *ifile) = 0;

    //设置输出文件路径
    virtual void setOutputFile(const char *ofile) = 0;

    //设置运行模式，0：VERBOSE模式，1：安静模式
    virtual void setRunMode(int mode = 0) = 0;
};
```

这里需要特别说明的是：

（1）RandomEventTarget 是定义的枚举类型（具体定义在 3.4 里说明），用来标识发送方和接收方。当调用 startTimer 和 stopTimer 时，该参数设为 SENDER；当调用 sendToNetworkLayer 方法时，该参数设为对方，即如果是发送方调用 sendToNetworkLayer 方法时发送数据报文，该参数设为 RECEIVER；如是接收方调用 sendToNetworkLayer 方法时发送确认报文，该参数设为 SENDER；当接收方调用 delivertoAppLayer 方法时，该参数设为 RECEIVER。

（2）startTimer 和 stopTimer 方法都需要设置 seqNum 参数，该参数应该是和该定时器相关的 Packet 序号。即使是 GBN 和简化版 TCP 这些协议，也要设置该参数（应为最早发出但未确认的 Packet 序号）。另外重新启动一个定时器前，一定要先关闭该定时器（要注意 seqNum 参数的一致性），否则模拟网络环境会提示“试图启动一个已启动的定时器”。

（3）setInputFile 和 setOutputFile 是增加的二个接口函数，设置输入文件和输出文件的路径。这是为了验证协议的正确性而添加的：模拟网络环境的发送方会读取文件，构造应用层的 Message，调用 RdtSender 的 send 方法将 Message 发送到接收方，接收方 RdtReceiver 再将正确收到的报文调用 delivertoAppLayer 方法交给接收方模拟网络环境的应用层，最后写入到输出文件。如果输入文件和输出文件内容一样，说明协议工作正确。测试用的输入文件请使用发布的 input.txt 文件，如果协议工作正确，程序会产生输出文件，并且输出文件和输入文件的内容一致。

（4）setRunMode 函数也是增加的接口函数，用于设置网络模拟环境的运行模式。如果设置 mode=0（也是该函数的缺省参数），则为 Verbose 模式，网络模拟环境会输出很多模拟环境的运行信息，可以帮助学生观察协议的工作过程，特别是协议实现有问题时，可以帮助分析协议出现的问题；如果设置为 mode=1，则为 Silence 模式，这是会关闭掉模拟环境输出的运行信息，而控制台只会输出学生协议实现代码里打印的信息。

3.4 其它定义

在 RandomEventEnum.h 头文件里定义了枚举类型，用来标识发送方和接收方。其定义为：

```
/* 定义随机事件的目标*/
enum RandomEventTarget {
```

```
    SENDER,                                 //数据发送方
    RECEIVER                                //数据接收方
};
```

在 Tool.h 头文件里，定义了可以使用的工具接口，其定义为：

```
struct Tool{
    /* 打印Packet的信息*/
    virtual void printPacket(const char * description, const Packet &packet) = 0;
    /*计算一个Packet的校验和*/
    virtual int calculateCheckSum(const Packet &packet) = 0;
    /*产生一个均匀分布的[0-1]间的随机数*/
    virtual double random() = 0;
};
```

其中 void printPacket(const char * description, const Packet &packet)函数可以用来打印调试信息，第一个参数为描述性字符串，第二个参数为要打印输出的 Packet；int calculateCheckSum(const Packet &packet)函数计算给定 Packet 的校验和；double random()函数是模拟网络环境所需的。这里要特别说明的是校验和的计算请调用 Tool 接口定义的方法。

在 Global.h 里声明了二个全局指针，分别指向实现了 Tool 接口和 NetworkService 接口的实例，学生代码中通过这二个指针调用工具接口提供的函数和模拟网络环境提供的函数。由于这二个指针没有封装在智能指针里，学生需要在 main 函数结束前 delete 这二个指针。

第四章停止等待协议（Rdt3.0）实现示例

4.1 开发环境配置

1：link 模拟网络环境静态库

模拟网络环境已经被编译成静态库的 lib 文件（文件名为 netsimlib.lib），因此学生的代码工程编译时需要 link 这个 lib 文件。以 Windows 平台下的 VS2017 为例，可以有很多办法 link 一个自定义的静态库 lib 文件，在 StopWait 示例工程中，是在 stdAfx.h 头文件里加上如下的编译预处理指令将静态库链接在一起：

```
#pragma comment (lib,"D:\\DotNetProject\\VS2017\\Rdt\\Debug\\netsimlib.lib")
```

在上面的编译预处理指令中，需要指明 netsimlib.lib 文件的路径。如果是在 Window 下使用其它开发环境，请各位同学自己查阅自己的开发工具的帮助文档，采用适当的办法 link 到 netsimlib.lib 文件。

模拟网络环境在 Linux 环境下也已经被编译成静态库的 lib 文件（文件名为 libnetsim.a），以 Linux 下 StopWait 示例工程为例，工程的目录结构如图 2 所示：

bin	2018/11/13 0:50	文件夹	
CMakeFiles	2018/11/13 0:50	文件夹	
include	2018/11/13 0:50	文件夹	
lib	2018/11/13 0:50	文件夹	
src	2018/11/13 0:50	文件夹	
cmake_install.cmake	2018/11/13 0:48	CMAKE 文件	2 KB
CMakeCache.txt	2018/11/13 0:48	TXT 文件	12 KB
CMakeLists.txt	2018/11/13 0:47	TXT 文件	1 KB
Makefile	2018/11/13 0:48	文件	7 KB

图 2 Linux 示例工程结构

其中 src 目录为.cpp 文件所在目录，include 为.h 文件所在目录，libnetsim.a 文件位于 lib 目录下，最终编译生成的可执行文件放在 bin 目录下。CMakeLists.txt 文件内容为：

```
cmake_minimum_required(VERSION 3.5)
PROJECT(stop_wait)

SET(CMAKE_C_COMPTLER GCC)
add_definitions(-std=c++11)
INCLUDE_DIRECTORIES(${PROJECT_SOURCE_DIR}/include)
aux_source_directory(${PROJECT_SOURCE_DIR}/src SRC_LIST)
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
ADD_EXECUTABLE(stop_wait ${SRC_LIST})
FIND_LIBRARY(NETSIM_LIB libnetsim.a ${PROJECT_SOURCE_DIR}/lib)
TARGET_LINK_LIBRARIES(stop_wait  ${NETSIM_LIB})
```

2：需要包含的头文件

在学生代码工程里，需要包含以下头文件：

- DataStructure.h
- Global.h
- NetworkService.h
- RandomEventEnum.h
- RdtReceiver.h
- RdtSender.h
- Tool.h

另外还需要如下常规的头文件：

- stdio.h
- string.h
- iostream

4.2 停止等待协议发送方实现

停止等待协议发送方是通过类 StopWaitRdtSender 实现，而 StopWaitRdtSender 继承了抽象类 RdtSender，具体定义如下：

```
class StopWaitRdtSender :public RdtSender
{
private:
    int expectSequenceNumberSend;          // 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
public:

    bool getWaitingState();
    //发送应用层下来的Message，由NetworkServiceSimulator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正确确认状态而拒绝发送Message，则返回false
    bool send(Message &message);

    //接受确认Ack，将被NetworkServiceSimulator调用
    void receive(Packet &ackPkt);

    //Timeout handler，将被NetworkServiceSimulator调用
    void timeoutHandler(int seqNum);
public:
    StopWaitRdtSender();
    virtual ~StopWaitRdtSender();
};
```

StopWaitRdtSender 类的函数实现具体为：

```
StopWaitRdtSender::StopWaitRdtSender():expectSequenceNumberSend(0),waitingState(false)
{
}
StopWaitRdtSender::~StopWaitRdtSender()
{
}
boolStopWaitRdtSender::getWaitingState() {
    return waitingState;
}
bool StopWaitRdtSender::send(Message &message) {
    if (this->waitingState) { //发送方处于等待确认状态
        return false;
    }

    this->packetWaitingAck.acknum = -1; //忽略该字段
    this->packetWaitingAck.seqnum = this->expectSequenceNumberSend;
    this->packetWaitingAck.checksum = 0;
    memcpy(this->packetWaitingAck.payload, message.data, sizeof(message.data));
    this->packetWaitingAck.checksum = pUtils->calculateCheckSum(this->packetWaitingAck);
    pUtils->printPacket("发送方发送报文", this->packetWaitingAck);

    //启动发送方定时器
    pns->startTimer(SENDER, Configuration::TIME_OUT,this->packetWaitingAck.seqnum);

    /调用模拟网络环境的sendToNetworkLayer，通过网络层发送到对方
    pns->sendToNetworkLayer(RECEIVER, this->packetWaitingAck);
    //进入等待状态
    this->waitingState = true;
    return true;
}

void StopWaitRdtSender::receive(Packet &ackPkt) {
    //如果发送方处于等待ack的状态，作如下处理；否则什么都不做
    if (this->waitingState == true) {
        //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->packetWaitingAck.seqnum) {
            //下一个发送序号在0-1之间切换
            this->expectSequenceNumberSend = 1 - this->expectSequenceNumberSend;
            this->waitingState = false;
```

```
            pUtils->printPacket("发送方正确收到确认", ackPkt);
            //关闭定时器
            pns->stopTimer(SENDER, this->packetWaitingAck.seqnum);
        }
        else {
            pUtils->printPacket("发送方没有正确收到确认，重发上次发送的报文",
                                                    this->packetWaitingAck);
            //首先关闭定时器
            pns->stopTimer(SENDER, this->packetWaitingAck.seqnum);
            //重新启动发送方定时器
            pns->startTimer(SENDER, Configuration::TIME_OUT, this->packetWaitingAck.seqnum);
            //重新发送数据包
            pns->sendToNetworkLayer(RECEIVER, this->packetWaitingAck);
        }
    }
}

voidStopWaitRdtSender::timeoutHandler(intseqNum) {
    //唯一一个定时器,无需考虑seqNum
    pUtils->printPacket("发送方定时器时间到，重发上次发送的报文", this->packetWaitingAck);
    //首先关闭定时器
    pns->stopTimer(SENDER,seqNum);
    //重新启动发送方定时器
    pns->startTimer(SENDER, Configuration::TIME_OUT,seqNum);
    //重新发送数据包
    pns->sendToNetworkLayer(RECEIVER, this->packetWaitingAck);
}
```

4.3 停止等待协议接收方实现

停止等待协议发送方是通过类 StopWaitRdtReceiver 实现，而 StopWaitRdtReceiver 继承了抽象类 RdtReceiver，具体定义如下：

```
class StopWaitRdtReceiver :publicRdtReceiver
{
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:
    StopWaitRdtReceiver();
    virtual ~StopWaitRdtReceiver();

public:
```

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

StopWaitRdtReceiver 类的函数实现具体为：

```
StopWaitRdtReceiver::StopWaitRdtReceiver():expectSequenceNumberRcvd(0)
{
    //初始状态下，上次发送的确认包的确认序号为-1，
    //使得当第一个接受的数据包出错时该确认报文的确认号为-1
    lastAckPkt.acknum = -1;
    lastAckPkt.checksum = 0;
    lastAckPkt.seqnum = -1;    //忽略该字段
    for(inti = 0; i<Configuration::PAYLOAD_SIZE;i++){
        lastAckPkt.payload[i] = '.';
    }
    lastAckPkt.checksum = pUtils->calculateCheckSum(lastAckPkt);
}
StopWaitRdtReceiver::~StopWaitRdtReceiver()
{
}

void StopWaitRdtReceiver::receive(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);
        pUtils->printPacket("接收方发送确认报文", 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
        pns->sendToNetworkLayer(SENDER, lastAckPkt);
        //接收序号在0-1之间切换
        this->expectSequenceNumberRcvd = 1 - this->expectSequenceNumberRcvd;
    }
    else {
        if (checkSum != packet.checksum) {
```

```
            pUtils->printPacket("接收方没有正确收到发送方的报文,数据校验错误", packet);
        }
        else {
            pUtils->printPacket("接收方没有正确收到发送方的报文,报文序号不对", packet);
        }
        pUtils->printPacket("接收方重新发送上次的确认报文", 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
        pns->sendToNetworkLayer(SENDER, lastAckPkt);
    }
}
```

4.4 启动模拟网络环境

模拟网络环境的启动在 main 函数里实现，具体代码为：

```
#include"stdafx.h"
#include"Global.h"
#include"RdtSender.h"
#include"RdtReceiver.h"
#include"StopWaitRdtSender.h"
#include"StopWaitRdtReceiver.h"


int main(intargc, char* argv[])
{
    //如果需要使用其它的Rdt协议，只需要实例化其他具体Rdt实现类的实例，
    //如GBNRdtSender和GBNRdtSeceiver
    RdtSender *ps = newStopWaitRdtSender();
    RdtReceiver * pr = newStopWaitRdtReceiver();
    pns->setRunMode(0);   //VERBOS模式
    //pns->setRunMode(1);   //安静模式
    pns->init();
    pns->setRtdSender(ps);
    pns->setRtdReceiver(pr);
    pns->setInputFile("C:\\Users\\crackryan\\Desktop\\input.txt");
    pns->setOutputFile("C:\\Users\\crackryan\\Desktop\\output.txt");
    pns->start();

    delete ps;
    delete pr;
    //指向唯一的工具类实例，只在main函数结束前delete
    delete pUtils;
    //指向唯一的模拟网络环境类实例，只在main函数结束前delete
    delete pns;
```

```
    return 0;
}
```

4.5 实验结果输出

为避免程序长时间运行，模拟网络环境当已发送的应用层 Message 达到某个门限时，程序会自动终止，模拟网络环境会输出调试信息帮助学生分析协议的执行过程，学生可以将控制台输出的结果重定向到文本文件里进行分析。图 3 为 StopWait 执行的结果截屏：

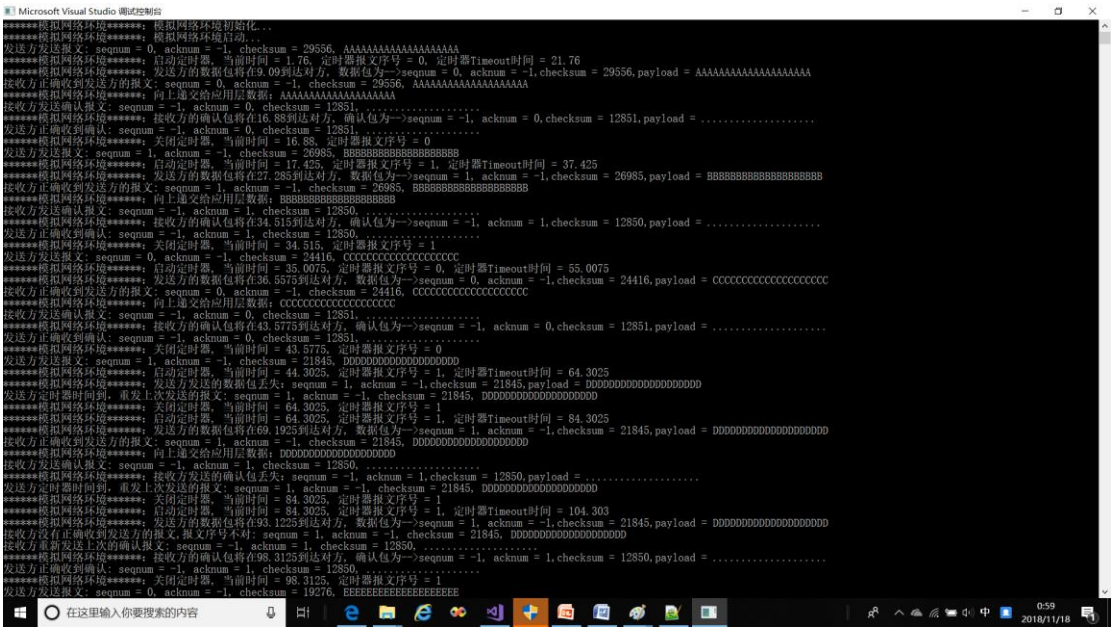

图 3 程序结果输出

程序输出的信息中，以“******”开始的行为模拟网络环境输出的信息。为了方便调试和分析，学生代码应该在合适的位置输出自己的信息。

为了更方便地验证 Rdt 协议工作的正确性，模拟网络环境的发送方会读取文件，构造应用层的 Message，调用 RdtSender 的 send 方法将 Message 发送到接收方，接收方 RdtReceiver 再将正确收到的报文调用 delivertoAppLayer 方法交给接收方模拟网络环境的应用层，最后写入到输出文件。如果输入文件和输出文件内容一样，说明协议工作正确。为了方便测试，在工程打包文件里有一个测试用的输入文件 input.txt 文件，学生可以将这个文件作为测试输入文件。

4.6 其他需要注意的问题

在实现具体 Rdt 协议时，需要考虑序号空间和窗口大小的问题。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），要注意窗口大小和编码报文序号的二进制位数之间的关系。更重要的是，在判断滑动窗口的位置和收到的确认报文序号关系时，不能简单按照教材上有限状态图里的伪代码那

样直接比较大小关系。建议将编码报文序号的二进制位数设为 3，窗口大小设为 4。

另外需要注意的是定时器的问题，关闭和启动定时器时一定要注意序号。另外重新启动一个定时器前一定要先关闭该定时器。

最后需要说明的时由于产生丢包和报文损坏都是利用随机事件产生的。因此需要多次运行，每次运行后比较二个文件内容是否相同，这样才能确保协议的正确性。