《计算机网络协议开发》实验报告

第四次实验
滑动窗口协议实验

姓名：元玉慧
学号：101220151
10 级 计算机 系 4 班
邮箱：njucsyyh@gmail.com
时间：2013/04/08

一、 实验目的

本次实验是在一个链路层的模拟现实环境中，用 C 语言来分别实现 1 比特滑动窗口协议，回退 N 帧协议和选择性重传协议，更深刻地理解滑动窗口协议

二、实验设计要点

滑动窗口的机制：

基本原理：

任意时刻，发送方维持了一个连续的允许发送的帧的序号，成为发送窗口；同时接收方也维持了一个连续的允许接收的帧的序号，称为接收窗口，发送窗口和接收窗口的序号的上下界不一定要一样，甚至大小可以不同，不同的协议的窗口的大小一般也是不同的。

发送方维持了一个已发送但是还未确认的帧的序号表，称为发送窗口，发送窗口的上界表示要发送的下一个帧的序号，下界表示未得到确认的帧的最小序号。窗口大小=上界-下界，大小可变；

发送方每发送一个帧，序号取上界值，上界加 1；

每收到一个确认序号等于发送窗口下界的响应帧，下界加 1；

若确认序号在发送窗口之内，则发送窗口下界连续加 1，直到发送窗口下界 = 确认序号+1

接收方有一个接收窗口，大小固定，但是不一定与发送窗口相同，接收窗口的上界表示允许接收的最大序号，下界表示希望接收的序号，接收窗口容纳允许接收的信息帧，落在窗口外的帧都被丢弃，序号等于下界的帧被正确接收，则产生一个响应帧，上界和下界都加 1 接收窗口的大小保持不变。

不同的滑动窗口协议：

1 比特滑动窗口协议：发送和接收窗口大小都是 1

回退 N 帧协议和选择性重传协议：发送窗口>1; 接收窗口>1;

-2-接口函数说明

【1】1 比特滑动窗口协议测试函数：

Int std_slide_window_stop_and_wait(char * pBuff, int buffersize, UINT8 messageType)

参数分析：

pBuff: 指向系统要发送或者接收到的内容，或者指向超时消息中超时帧的序号内容。当 pBuff 指向的是系统要发送或者接收的帧内时，其指向的数据为网络字节序。

buffersize：pBuffer 表示内容的长度。

messageType：传入的消息类型，可以分为如下几种情况：

MSG_TYPE_TIMEOUT 某个帧超时

MSG_TYPE_SEND　　系统要发送一个帧

MSG_TYPE_RECEIVE　　系统要接收到的一个帧的 ACK

MSG_TYPE_TIMEOUT 消息，pBuffer 指向的数据的前四个字节为超时帧的序号，以 UINT32 类型存储，在与帧中的序号进行比较时，需要注意字节序，进行必要的转换。

MSG_TYPE_SEND 和 MSG_TYPE_RECEIVE 类型的消息，pBuffer 指向数据的结构如下代码中 frame 结构的定义：

```
typedef enum { data, ack , nak }  frame_kind;
typedef struct frame
{
    framekind kind   //帧类型
    unsigned int seq;   //序列号
    unsigned int ack;   //确认号
    unsigned char data[100];   //数据
}
```

【2】回退 N 帧协议测试函数：

Int std_slide_window_back_n_frame(char * pBuff, int buffersize, UINT8 messageType)

【3】选择性重传协议测试函数：

Int std_slide_window_choice_frame_resend (char * pBuff, int buffersize, UINT8 messageType)

【4】系统系统的接口函数说明：

发送函数

Extern void SendFRAMEPacket(unsigned char *pData, unsigned int len);

参数说明：

pData:指向要发送的帧的内容的指针，其指向的帧内容应以网络字节序存储。

Len：要发送的帧的长度。

实验的具体测试情况

【1】 发送端发送帧时，会调用学生测试函数，并置参数 messageType 为 MSG_TYPE_SEND；测试函数会将该帧缓存，存入发送队列中。若发送窗口还未打开到规定限度，则打开一个窗口，并将调用 SendFRAMEPacket 函数将该帧发送。若发送窗口已经达到了打开限度，则直接返回，相当于直接进入了等待状态。

【2】 当发送端收到接收端的 ACK 之后，会调用学生的测试函数，并置该 ACK 对应的窗口关闭，由于关闭了窗口，等待发送的帧可以进入窗口并发送，因此，此时若发送

队列中存在等待发送的帧，应该将一个等待发送的帧发送并打开一个新的窗口

【3】 发送端每发送一个帧，系统都会为它创建一个定时器，当成功收到 ACK 帧后，定时器会被取消，若某个帧在定时器超时后仍未收到回复 ACK，系统会调用测试函数，并置参数为 MSG_TYPE_TIMEOUT，告知测试函数某帧超时，测试函数应该根据帧序号将该帧以及后面发送过来的帧重新发送（对选择性重传协议而言，是只重新发送超时的帧）

【4】 发送端收到接收端的帧类型为 NAK 后，会调用学生的测试函数，并置参数 messageType 为 MSG_TYPE_RECEIVE，测试函数应该检查 NAK 的值后，根据帧的序号将该帧以及后面发送过来的帧重新发送（对于选择性重传协议，只重新发送出错的帧）

设计简化分析：

在实现滑动窗口协议的 3 个接口函数的时候，不需要设置源，目的地址，以及考虑超时定时器的设计。

在编写滑动窗口协议的 3 个接口函数时，代码中构建的缓冲区可以不需要考虑帧发送后的清理问题。

三、实验内容

定义发送的窗口和完整的帧的结构体：

```
21  //帧头
22  typedef struct frame_head
23  {
24      frame_kind kind;     //帧类型
25      int seq;             //序列号
26      int ack;             //确认号
27      unsigned char data[100];     //数据
28  };
29
30  //帧
31  typedef struct frame
32  {
33      frame_head head; //帧头
34      int size;        //数据的大小
35  };
36
37  //发送窗口
38  typedef struct Windows
39  {
40      frame *pframe;    //帧指针
41      int size;        //内容的长度
42  };
43
```

用双向队列来作为发送消息的缓冲区

```
45  deque <Windows> deque1;
46  deque <Windows> deque2;
47  deque <Windows> deque3;
48  bool IsFull = false;
```

（核心代码）

【1】1 比特滑动窗口协议测试函数设计：

处理发送消息

```
if(MSG_TYPE_SEND == messageType){
    buff.pframe = new frame;
    *buff.pframe = *(frame *)pBuffer;
    buff.size = bufferSize;
    deque1.push_back(buff);
    if(!IsFull)
    {
        buff = deque1.back();
        //调用发送帧函数
        SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
        //发送窗口已满
        IsFull = true;
    }
}
```

处理接收的消息：

```
if(MSG_TYPE_RECEIVE == messageType){
    buff = deque1.front();
    ack = ntohl(((frame *)pBuffer)->head.ack);
    if(ntohl(buff.pframe->head.seq) == ack)
        //if(*buff.pframe.head.seq == ack)
    {
        deque1.pop_front();
        if(deque1.size() != 0 )
        {
            buff = deque1.front();
            SendFRAMEPacket(((unsigned char *)buff.pframe), buff.size);
        }
        else
        {
            //设置发送窗口为未满
            IsFull = false;
        }
    }
}
```

处理超时的消息：

```
//超时的处理
if(MSG_TYPE_TIMEOUT == messageType){
    num = ntohl(*(int *)pBuffer);
    buff = deque1.front();
    if(num == buff.pframe->head.seq)
    {
        //调用发送帧函数
        SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
    }
}
```

【2】回退 N 帧协议测试函数设计：

处理发送消息

```
if(MSG_TYPE_SEND == messageType){
    buff.pframe = new frame;
    (*buff.pframe) = *(frame *)pBuffer;
    buff.size = bufferSize;
    deque2.push_back(buff);
    if(Count2 < WINDOW_SIZE_BACK_N_FRAME)
    {
        buff = deque2.back();
        SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
        Count2++;
    }
 }
```

处理接收消息

```
if(MSG_TYPE_RECEIVE == messageType){
        ACK = ntohl(((frame *)pBuffer)->head.ack);
        for(i = 0; i < WINDOW_SIZE_BACK_N_FRAME && i < deque2.size(); i++)
        {
            buff = deque2[i];
            if(ACK == ntohl((*buff.pframe).head.seq))
            {
                break;
            }
        }
        if(i < deque2.size() && i < WINDOW_SIZE_BACK_N_FRAME)
        {
            for (j = 0; j <= i; j++)
            {
                deque2.pop_front();
                Count2--;
            }
        }
        for (; Count2 < WINDOW_SIZE_BACK_N_FRAME && Count2 < deque2.size(); Count2++)
        {
            buff = deque2[Count2];
            SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
        }
  }
```

处理超时消息

```
//超时的处理
if(MSG_TYPE_TIMEOUT == messageType){
    //测试后发现，服务器采用的重发策略是将窗口中的帧全部重发
    for (i = 0; i < WINDOW_SIZE_BACK_N_FRAME && i < deque2.size(); i++)
    {
        buff = deque2[i];
        SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
    }
}
return 0;
```

【3】选择性重传协议测试函数设计：

处理发送消息

```
if(MSG_TYPE_SEND == messageType){
    buff.pframe = new frame;
    *buff.pframe = *(frame *)pBuffer;
    buff.size = bufferSize;
    deque3.push_back(buff);
    if(Count3 < WINDOW_SIZE_CHOICE_FRAME_RESEND)
    {
        //获得发送队列中最晚压入的帧
        buff = deque3.back();
        SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
        Count3++;
    }
}
```

处理接收消息

```
//接收帧的处理
if(MSG_TYPE_RECEIVE == messageType){
    int frame_kind = ntohl(((frame *)pBuffer)->head.kind);
    //如果帧损坏，那么把出错的帧重新发送一次
    if(frame_kind == nak)
    {
        NAK = ntohl(((frame *)pBuffer)->head.ack);
        for(i = 0; i < WINDOW_SIZE_CHOICE_FRAME_RESEND && i < deque3.size(); i ++)
        {
            if(NAK == ntohl((*(deque3[i].pframe)).head.seq))
            {
                SendFRAMEPacket((unsigned char *)(deque3[i].pframe), deque3[i].size);
                break;
            }
        }
    }
```

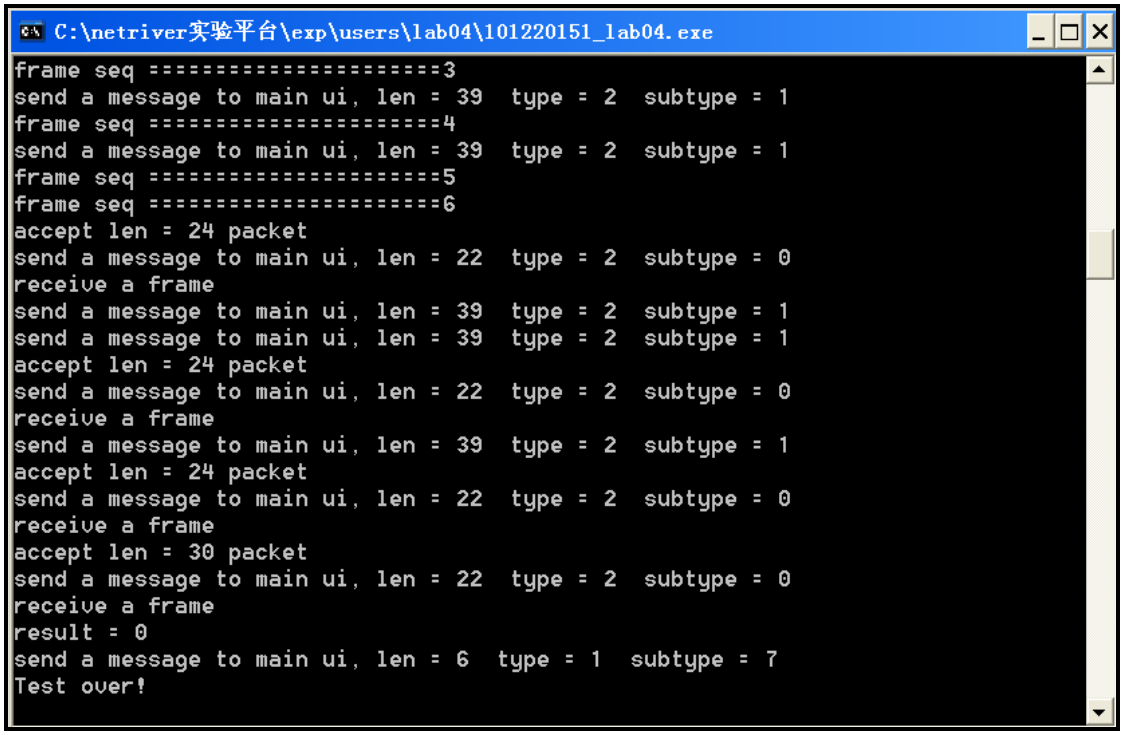
```
    else
    {
        ACK = ntohl(((frame *)pBuffer)->head.ack);
        for(i = 0; i < WINDOW_SIZE_CHOICE_FRAME_RESEND && i < deque3.size(); i++)
        {
            buff = deque3[i];
            if(ACK == ntohl((*buff.pframe).head.seq))
            {
                break;
            }
        }
        if(i < deque3.size() && i < WINDOW_SIZE_CHOICE_FRAME_RESEND)
        {
            for (j = 0; j <= i; j++)
            {
                deque3.pop_front();
                Count3--;
            }
        }
        //获取当前计数器的值
        //j = Count3;
        //将发送队列中可以发送的帧全部发送
        for (;Count3 < deque3.size() && Count3 < WINDOW_SIZE_CHOICE_FRAME_RESEND; Count3++)
        {
            buff = deque3[Count3];
            //调用发送函数
            SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
        }
    }
}
```

处理超时消息

```
if(MSG_TYPE_TIMEOUT == messageType){
    //将一个32位数由网络字节顺序转换为主机字节顺序
    num = ntohl(*(unsigned int *)pBuffer);
    //通过循环找到超时的帧的序列号
    for (i = 0; i < deque3.size() && i < WINDOW_SIZE_BACK_N_FRAME; i++)
    {
        buff = deque3[i];
        if ((*buff.pframe).head.seq == num)
            break;
    }
    buff = deque3[i];
    SendFRAMEPacket((unsigned char *)(buff.pframe), buff.size);
}
```

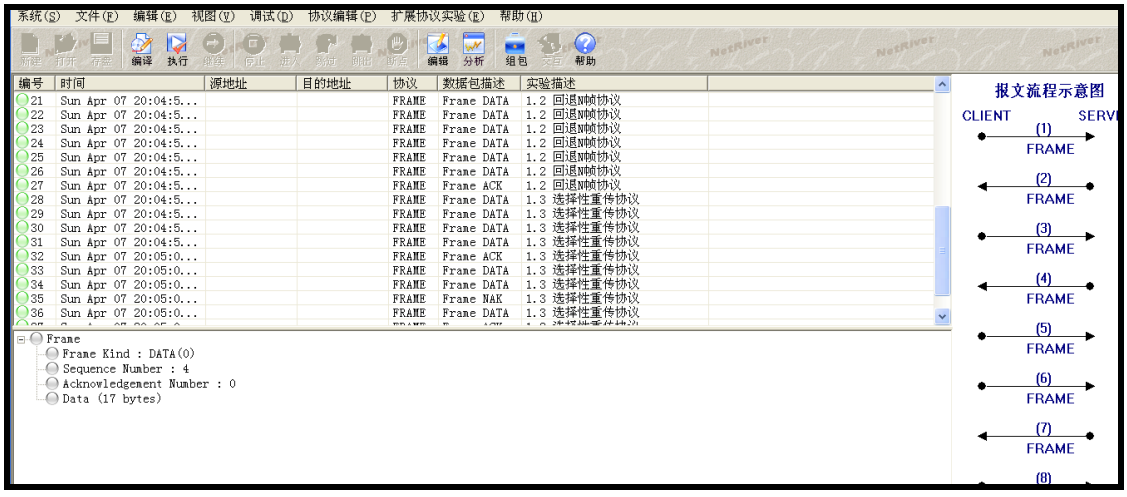
实验的设计思路主要是参考了书上的实验设计流程以及网上的一些设计技巧和方法，主要利用了 C++提供的<deque>的数据结构来实现的。

四、实验结果


C:\netriver实验平台\exp\users\lab04\101220151_lab04.exe
frame seq =====================3
send a message to main ui, len = 39 type = 2 subtype = 1
frame seq =====================4
send a message to main ui, len = 39 type = 2 subtype = 1
frame seq =====================5
frame seq =====================6
accept len = 24 packet
send a message to main ui, len = 22 type = 2 subtype = 0
receive a frame
send a message to main ui, len = 39 type = 2 subtype = 1
send a message to main ui, len = 39 type = 2 subtype = 1
accept len = 24 packet
send a message to main ui, len = 22 type = 2 subtype = 0
receive a frame
send a message to main ui, len = 39 type = 2 subtype = 1
accept len = 24 packet
send a message to main ui, len = 22 type = 2 subtype = 0
receive a frame
accept len = 30 packet
send a message to main ui, len = 22 type = 2 subtype = 0
receive a frame
result = 0
send a message to main ui, len = 6 type = 1 subtype = 7
Test over!


/*
* lab04.cpp
*
* Created on: 2013-4-4
* Author: njucsyyh
*/
#include "sysinclude.h"
#include <deque>
using namespace std;
extern void sendFRAMEPacket(unsigned char* pD
#define WINDOW_SIZE_STOP_WAIT 1
#define WINDOW_SIZE_BACK_N_FRAME 4
程序结束
测试结
1 滑动窗口协议实验
1.1 1比特滑动窗口协议 -- 成功
1.2 回退N帧协议 -- 成功
1.3 选择性重传协议 -- 成功
是否提交测试结果到服务器?
提交
取消
OK!

报文分析部分：


报文流程示意图
CLIENT
(1) FRAME
(2) FRAME
(3) FRAME
(4) FRAME
(5) FRAME
(6) FRAME
(7) FRAME
Frame
Frame Kind : DATA(0)
Sequence Number : 4
Acknowledgement Number : 0
Data (17 bytes)

五、思考问题

-1-滑动窗口协议的主要功能：

滑动窗口协议是 TCP 使用的一种流量控制方法，该协议允许发送方在停止并等待确认前可以连续的发送多个分组，由于发送方不必没法一个分组就停下来等待确认，因此该协议可以加速数据的传输。

TCP 的滑动窗口用来暂存 2 台计算机之间要传输的数据分组，每台运行 TCP 协议的计算机有 2 个滑动窗口：一个用于数据发送；另一个用于数据接收。

发送端待数据分组在缓冲区排队等待送出，被滑动窗口匡茹的分组，是可以在未收到确认的情况下最多发送的部分，滑动窗口左端标志 X 的分组，是已经被接收端确认的分组，随着新的确认到来，窗口不断向右滑动。

-2-回退 N 帧协议与 1 比特滑动窗口协议相比有何优点？

1 比特停等协议实现虽然简单，但是效率太低，回退 N 帧协议发送窗口大小为 N，接收方窗口大小为 1，发送方一次可以发送 N 帧数据，而没有必要等待确认一帧后再进行发送，所以大大提高了效率

-3-回退 N 帧协议有什么缺点，如何改进？

如果网络情况不理想的话，回退 N 帧协议可能会导致多帧重发，回退 N 帧协议不一定优于 1 比特滑动窗口协议，同时接收方在发送错误帧之后就不再接收后续的帧，即使是正确到达的帧。

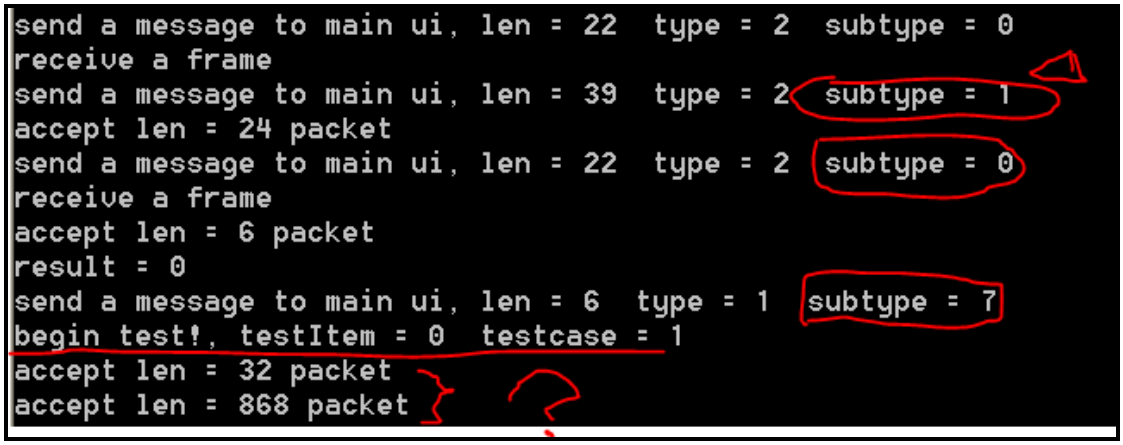
解决的方法是：

当接收方发现错误之后，仍然可以继续接收送来的正确的帧，存放在缓冲区中，同时要求发送发重新发送传送错误的那一帧，一旦受到重新传来的帧之后，就可以和原来已存于缓冲区中的其余帧一并按照正确的顺序提交给高层，也就是类似于选择性重传协议。

-4-试分析数据链路层的滑动窗口协议和传输层的滑动窗口协议的不同。

2 者的工作原理是完全相同过的，传输层滑动窗口协议的目的是实现端到端节点之间实现流量控制，而数据链路层则是为了实现相邻节点之间的流量控制，

-5-试分析点击“执行”后弹出的 DOS 窗口中出现的一些列字符串的含义。



其他字段都方便理解，通过分析，data 域占用了 17 个字节，分析可以知道
subtype=1 时，表示 data 类型的数据包；
subtype=0 时，表示 ack 类型的数据包；
subtype=7 时，表示 RTT 类型的数据包；(重新连接)
至于发送的
Accept len 为什么是 32 还有 868 具体是怎么算出来的，不是很清楚！

-6-你实现的回退 N 帧协议和选择性重传协议代码能解决响应报文乱序到达的问题吗？
不能处理乱序和重复的报文。

-7-在处理超时重发是，能否不管系统提供的重发帧序号，直接将缓冲区的第一个帧重发。（就是将窗口中所有的帧都发送）
可以 ,通过此次实验发现测试服务器就是采用了将缓冲区中所有的帧都重发的策略。只不过这样更加降低了传送的效率。

六、实验中遇到的问题

【1】在刚开始设计的时候不清楚可以使用 C++的<queue>库，本来想自己设计队列的链表数据结构，后来去网上搜集资料，查到一些实现思路，然后才发现实验平台可以使用<deque>
库。发现实验的实现容易了很多，可以使用 deque 结构来维护窗口和等待发送的窗口。并可以方便的删除已经得到确认的窗口。

【2】在设计测试回退 N 帧的协议时候发现了测试都是服务器超时后来对发送的

报文进行了分析观察，发现了一些问题，后来与同学讨论后，发现服务器默认如果发生超时的话，需要对滑动窗口中的个窗口内容全部重新发送。

【3】在测试的时候发现了服务器可能存在一些问题，比如设置的时候发现，登上服务器很久一段时间后，再次测试会发现第二项和第三项连接服务器超时。。。后来重新启动 netriver 软件后重新选择编译测试，一切 OK！可能跟我的实验环境有关吧，我是 win7 的 64 位系统，需要安装 XP 虚拟机，可能与虚拟环境有关吧。。

七、 实验的启示/意见和建议

通过此次实验，主要熟悉了 NetRiver 平台，并且通过实验加深了对滑动窗口协议的理解。

此次实验大约花费了 ***12+***小时吧！包括写代码+调代码+写报告