# 数据通信Lab Chapter3


## 黄奥成191180048 通信工程


## 1、数据结构

```
//Datagram A
typedef struct sequence
{
    int base;
    int nextseqnum;

}sequence;
sequence seq = {0,0};
msg sendbuffer[N_sendbuffer];


//Datagram B
pkt recvbuffer;
int recv_base = 0;

```

由于题目中提到，只需要20的序列，因此这里定义`sendbuffer`为一个较大的msg数组。如果为了节省空间，因为GBN只要求窗口N=8的缓存。因此完全可以只消耗N的数组，而使用下标`sendbuffer[seq.base % N]`到`sendbuffer[seq.nextseqnum % N]`这个非连续的数组来缓存正在发送的窗口的包。

来详细解释一下每一个参数的含义：

- `seq.base`正在发送的窗口中的首个序号
- `seq.nextseqnum`正在发送的窗口的后一个序号（不在窗口中）
- `recv_base`接收方希望收到的下一个序号

## 2、函数实现

### 2.1 检验和

```
void evaluate_checksum(pkt *packet){

    int checksum;
    checksum = packet->seqnum + packet->acknum;
    for (int i = 0; i < 20; i++)
        checksum = checksum + (int)(packet->payload[i]);
    checksum = 0 - checksum;
    packet->checksum = checksum;

}

bool Corrupted(pkt packet){
```

```
    int checksum = packet.checksum;
    evaluate_checksum(&packet);
    return packet.checksum != checksum;

}
```

`evaluate_checksum`计算检验和并将传入的packet的检验和修改。`Corrupted`比较检验和前后是否一直。这里有一个小细节，在编写程序的时候我思考了一下，如果说`acknum`和`seqnum`能在传输中corrupt，那么检验和是否能corrupt？因此我一开始的想法是，在发送方A接收到B的ACK报文时，将其检验和与本地缓存的检验和对比。但是RTFS(Read the Fxcking Source)

```
/* simulate corruption: */
    if (jimsrand() < corruptprob) {
        ncorrupt++;
        if ((x = jimsrand()) < .75)
            mypktptr->payload[0] = 'Z'; /* corrupt payload */
        else if (x < .875)
            mypktptr->seqnum = 999999;
        else
            mypktptr->acknum = 999999;
        if (TRACE > 0)
            printf("          TOLAYER3: packet being corrupted\n");
    }
```

可以发现，程序设定的检验和在传输过程中不会错误。

### 2.2 制作包

```
pkt make_pkt(int seqnum, int acknum,
             char payload[20]){

    pkt packet = {seqnum,acknum,0};
    memcpy(packet.payload,payload,20);

    evaluate_checksum(&packet);

    return packet;
}
void set_buffer(msg message){

    memcpy(sendbuffer[seq.nextseqnum].data,message.data,20);

}

```

提高代码复用性和可读性，btw，写了一个星期python回来用c好不习惯😂。而且是c不是c++有点头疼

### 2.3 A_output

逻辑比较简单，判断`seq.nextseqnum-seq.base`是否大于窗口，如果不是则创建包并且发送

```
void A_output(struct msg message)
{
    if(seq.nextseqnum>=nsimmax){
            printf("The buffer is full,and the program will be stopped!\n");
            exit(0);
```

```
        }
    //若大于窗口大小直接丢弃包
    else{
        if (seq.nextseqnum - seq.base < N){
            printf("A send seqnum %d to layer3",seq.nextseqnum);
            pkt output_packet = make_pkt(seq.nextseqnum,0,message.data);
            set_buffer(message);
            seq.nextseqnum ++;

            tolayer3(A,output_packet);
            stoptimer(A);
            starttimer(A,20.0);
        }
        else{
            printf("The Windows is already full!\n");
            return;
        }
    }
}
```

### 2.4 A_input

这个函数真的是头疼，一开始我想按照自己的想法写而不按照作业要求写，结果处处碰壁。

需要注意两个点，也是我debug时候遇到的

- 不需要按照ACK的顺序接收，而且接收到最大的ACK就表示之前的已经ACK
- 存在`seq.base==seq.nextseqnum`，这个时候不需要开启定时器(这就表示窗口内没有待发送的包)

这两个bug恶心了我小几个小时，第一个是我忽视了GBN也是rdt3.0的改良，第二个确实没想到😂最后通过输出日志一条一条看才debug出来

```
void A_input(struct pkt packet)
{
    //如果检验和相同
    if(!Corrupted(packet)){
        if(packet.seqnum == seq.base - 1)
        {
            printf("The packet sent to B is corrupted, Please send again!\n");
            pkt output_packet = make_pkt(seq.base,0,sendbuffer[seq.base].data);
            tolayer3(A,output_packet);
        }
        else if (packet.seqnum >= seq.base)
        {
            printf("A recv ACK%d from layer3\n",packet.seqnum);
            stoptimer(A);
            seq.base = packet.seqnum + 1;

            if(seq.base<seq.nextseqnum)
                starttimer(A,InterruptTime);
        }
    }
    else{
            printf("The recv packet corrupted!");
    }
}
```

### 2.5 B_input

input函数的逻辑也是简单到不行。但是一开始我的`A_input`函数写错了，即必须顺序接收每一个ACK，这就导致我`B_input`的逻辑一开始写错。而在更早的时候，其实我是注意到书本上的这句话

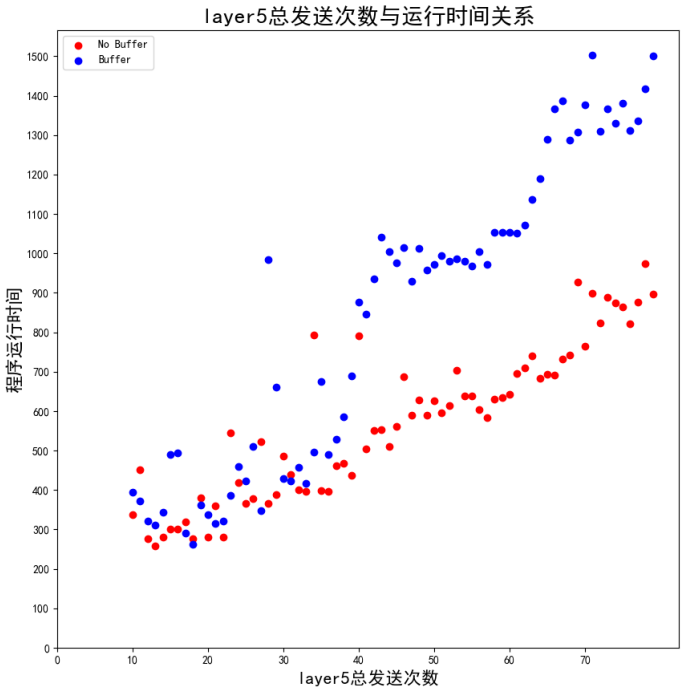
> 层。假定现在期望接收分组 $n$，而分组 $n+1$ 却到了。因为数据必须按序交付，接收方可能缓存（保存）分组 $n+1$，然后，在它收到并交付分组 $n$ 后，再将该分组交付到上层。然

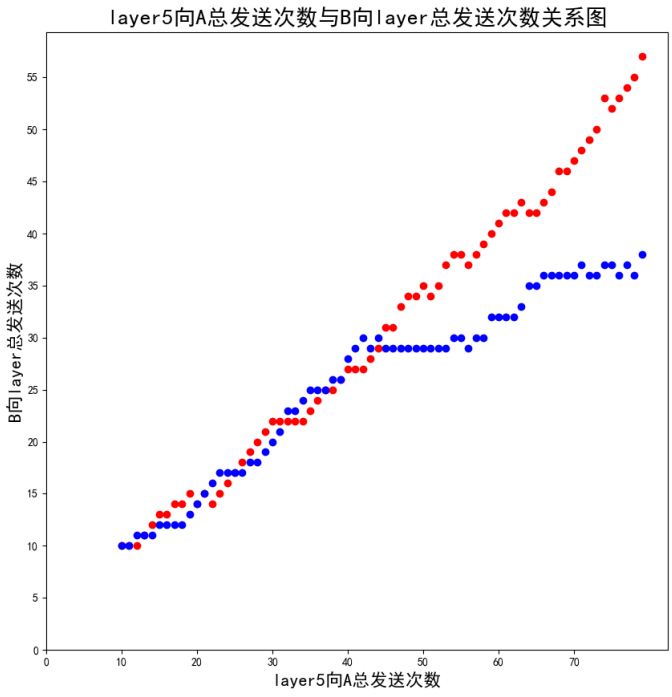
因此我一开始是在接收端设置了一个大小为N的buffer，然后在收到大于期望接收分组n的时候，将缓存窗口内的分组，也即[n+1,n+N-1]。而这里为了简便（偷懒），直接使用一个超级大的数组`buffer`。事实上，使用大小为窗口大小N=8的buffer同样能达到这个效果(在datagram部分有解释)，但是为了偷懒就懒得写了。

```
void B_input(struct pkt packet)
{
    if(!Corrupted(packet)){
        printf("recv_base%d  packet_seq %d\n",recv_base,packet.seqnum);

        if(packet.seqnum == recv_base){

            recv_base++;
            recvbuffer = packet;
            send_ack(B);

            tolayer5(B,packet.payload);
            printf("B send packet %d to layer5\n",packet.seqnum);
#ifdef BUFFER
            for(int i = recv_base;i<recv_base +N;i++){
                if(i ==buffer[i].seqnum){
                    recv_base++;
                    recvbuffer = buffer[i];
                    send_ack(B);
                    tolayer5(B,packet.payload);
                    printf("B send packet %d to layer5\n",packet.seqnum);
                }
                else{
                    break;
                }
            }
#endif

        }

        else if(packet.seqnum > recv_base ){
            buffer[packet.seqnum] =packet;
            send_ack(B);
            printf("B has not recieved %d yet\n", packet.seqnum );
        }
        else{
        }
    }
    else{
        //如果包受损，则向A发送最后一个收到的序列
        send_ack(B);
        //starttimer(B,InterruptTime);
```

```
43          printf("B's packet corrupted \n");
44
45      }
46
47  }
```

阅读以下代码可以发现，在中间部分有一个预编译经常用到的`#ifdef`，这里是为了测试有buffer和无buffer的系统。可以发现，buffer有效的提高了运行效率以及包发送率



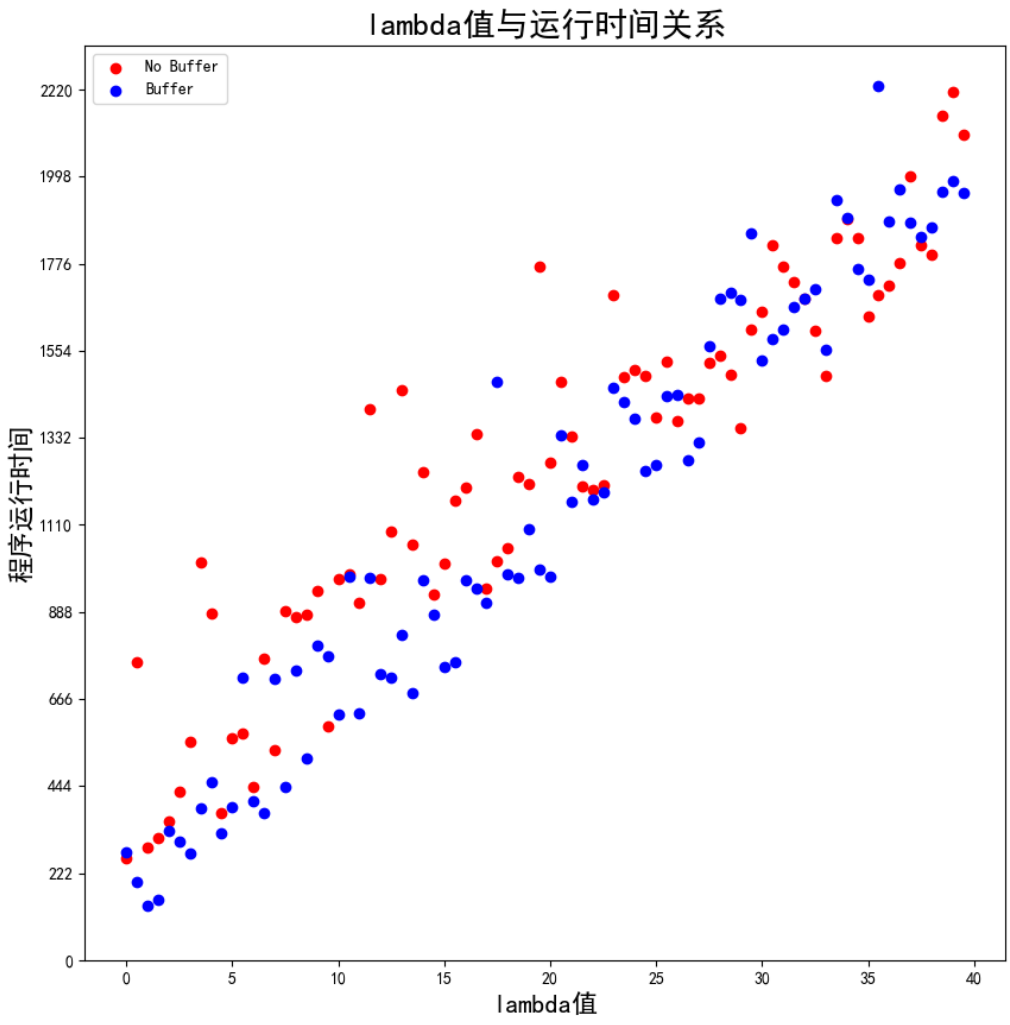



### 2.6 Time_interrupt

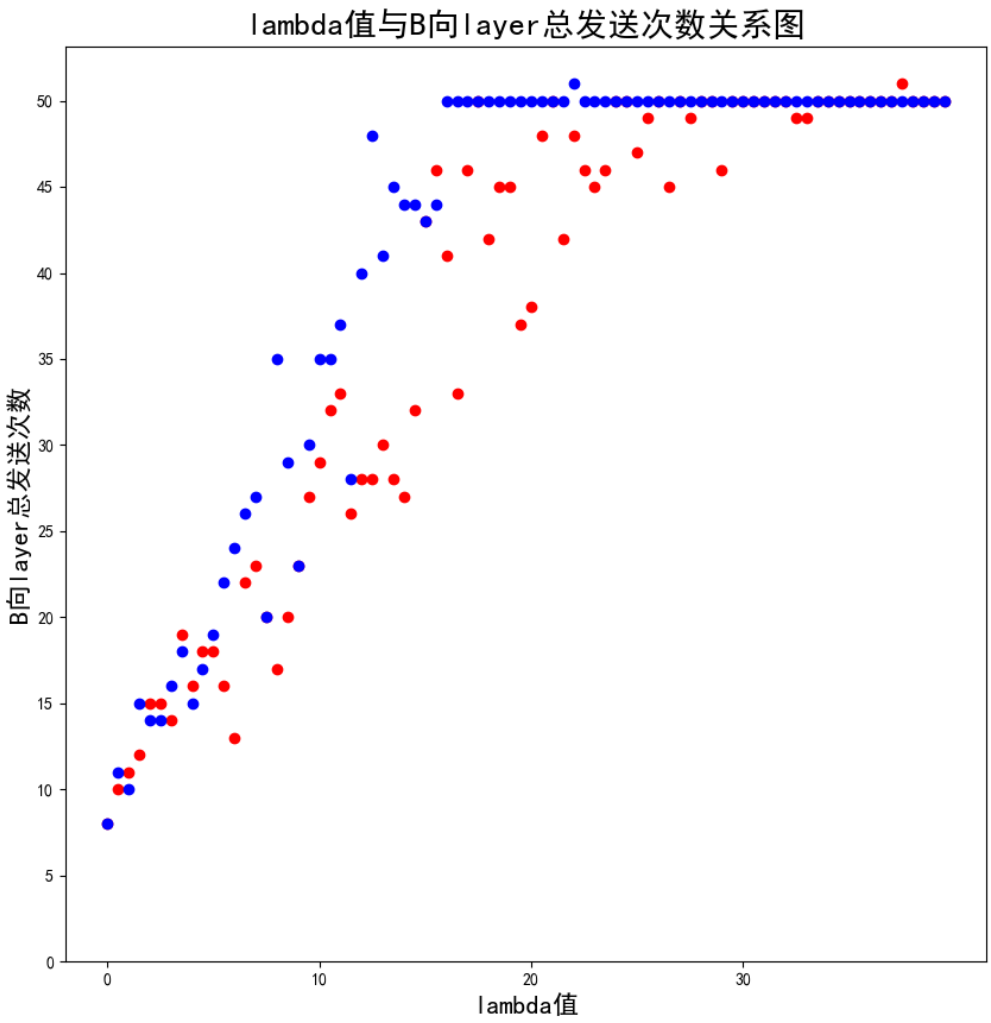
```
void A_timerinterrupt(void)
{

    printf("time interrupt,and the base now is %d and next is
%d\n",seq.base,seq.nextseqnum);
    starttimer(A,InterruptTime);
    for(int i =seq.base;i<seq.nextseqnum;i++){
        pkt packet = make_pkt(i,0,sendbuffer[i].data);
        printf("A send seqnum %d to layer3\n",i);
        tolayer3(A,packet);
    }

}
```

## 3、 实验运行

可以发现在发送数量相同时lambda值也即包发送的间隔对程序运行时间成正比关系(这不是废话)



在lambda值较低时（也即在10附近以下），发送到达率较低。




最后附上脚本与绘图

```
import os  #for exit
import time
from matplotlib.pyplot import MultipleLocator
import matplotlib.pyplot as plt

os.system("gcc prog2.c")

rate = []
N=80
os.system('rm -r output.txt')
time = 0
for i in range(80):
        time = i*0.5
        os.system('./a.out 50 0.2 0.2 {} 2'.format(time))
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False  # 用来正常显示负号
rate = []
N=80
with open('output3_nobuffer.txt','r') as f:
    while(1):
```

```
        fstr = f.readline()
        if(fstr == ''):
            break
        recv_base = fstr.split()[0]
        t = fstr.split()[1]
        rate.append([recv_base,t])
print(rate)
x=[i*0.5 for i in range(N)]
y= []
z= []
for i in range(len(rate)):
    y.append(int(rate[i][0]))
    z.append(float(rate[i][1]))


rate2 = []
with open('output3_buffer.txt','r') as f:
    while(1):
        fstr = f.readline()
        if(fstr == ''):
            break
        recv_base = fstr.split()[0]
        t = fstr.split()[1]
        rate2.append([recv_base,t])

x2=[i*0.5 for i in range(N)]
y2= []
z2= []
for i in range(len(rate2)):
    y2.append(int(rate2[i][0]))
    z2.append(float(rate2[i][1]))


plt.figure(1,figsize=(10, 10), dpi=100)
plt.scatter(x, y, c='red',label='No Buffer')
plt.scatter(x, y2, c='blue',label='Buffer')

plt.xticks(range(0, 40, 10))
plt.yticks(range(0, int(max(y)), int(int(max(y))/10)))
plt.xlabel("lambda值", fontdict={'size': 16})
plt.ylabel("B向layer总发送次数", fontdict={'size': 16})
plt.title("lambda值与B向layer总发送次数关系图", fontdict={'size': 20})


plt.figure(2,figsize=(10, 10), dpi=100)

plt.scatter(x, z, c='red',label='No Buffer')
plt.scatter(x, z2, c='blue',label='Buffer')

plt.yticks(range(0, int(max(z2)), int(int(max(z2))/10)))
plt.xlabel("lambda值", fontdict={'size': 16})
plt.ylabel("程序运行时间", fontdict={'size': 16})
plt.title("lambda值与运行时间关系", fontdict={'size': 20})
plt.legend(loc='best')

plt.show()
```