实验 3：基于 UDP 服务设计可靠传输协议并编程实现（任务 3-2）

姓　名: 杨科迪　　学　　号: 1813828
专　业: 计算机科学与技术　　指导老师: 徐敬东
实验时间：2020 年 12 月 11 日　　实验地点：实验楼 A 区 204

目录

1 实验目的

1. 实现基于滑动窗口的流量控制机制
2. 采用固定窗口大小
3. 支持累计确认

2 实验原理

1. 本实现基于 GBN。GBN 发送端 FSM 如图1所示，有以下 4 个事件：
   (a) 应用层发送数据: 如果 nextseqnum < base + N, 即滑动窗口中还有空余位置，则将数据封装后发送至接收端。如果发送前缓冲区为空则开启定时器；如果滑动窗口已满，则拒绝应用层发送数据。
   (b) 收到的报文损坏: 将报文丢弃
   (c) 收到的 ack 报文未损坏: base = acknum + 1，如果滑动窗口变为空，则停止定时器，否则重启定时器。
   (d) 超时: 将 [base, nextseqnum) 区间内的报文重传给接收端

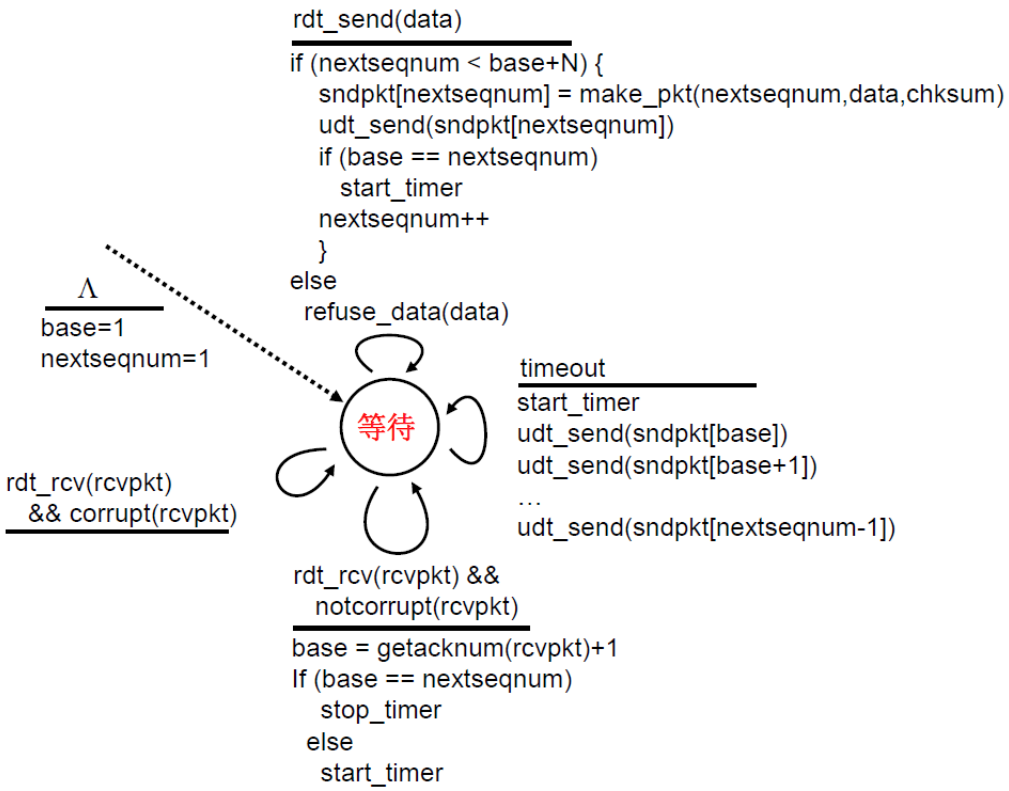

图 1: GBN 发送端 FSM

2. GBN 接收端 FSM 如图2所示，有以下两个事件：
   (a) 接收的报文未损坏并且按序: 向发送端发送 ack，确认号为正确接收的最高序号，然后将数据交付给应用层。
   (b) 接受的报文损坏或失序: 向发送端发送 ack，确认号为正确接收的最高序号。

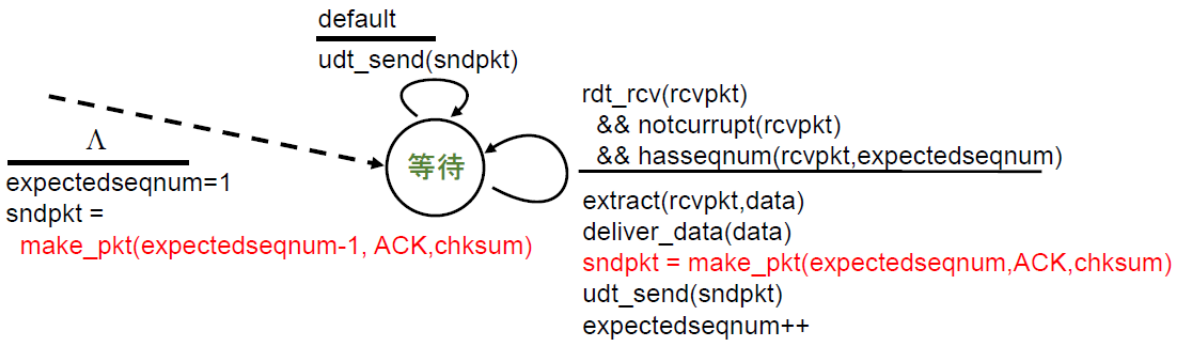

图 2: GBN 接收端 FSM

3. 本实验在 GBN 的基础上进行了如下改变：
   (a) 发送缓冲区

   GBN 中若滑动窗口已满，则应用层不能发送数据。本实验应用层将数据写入发送缓冲区中，滑动窗口在发送缓冲区滑动。如图3所示，[base, nextseqnum) 区间内为已发送未确认的数据报，[nextseqnum, nextwrite) 为应用层写入但未发送的数据，则有以下限制条件：

$$base <= nextseqnum <= nextwrite \tag{1}$$

$$nextseqnum < base + N \tag{2}$$

$$nextwrite < base + len(sendbuf) \tag{3}$$

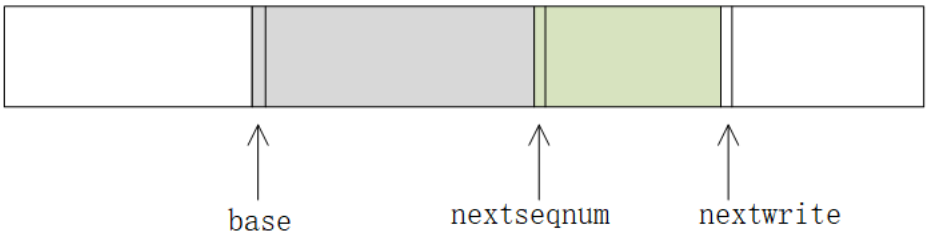

图 3: 发送缓冲区

   (b) 接收缓冲区

   接收端存在接收缓冲区，接收到数据后将数据缓存到接收缓冲区中，应用层从接收缓冲区中读取数据。如图4所示，nextread 为应用层下一次要读取的数据，nextrecv 为传输层下一次接收到的数据

存放的位置，[nextread, nextrecv) 为接收缓冲区按序收到的分组。有以下限制条件：

$$nextread <= nextrecv \tag{4}$$

$$nextrecv < nextread + len(recvbuf) \tag{5}$$

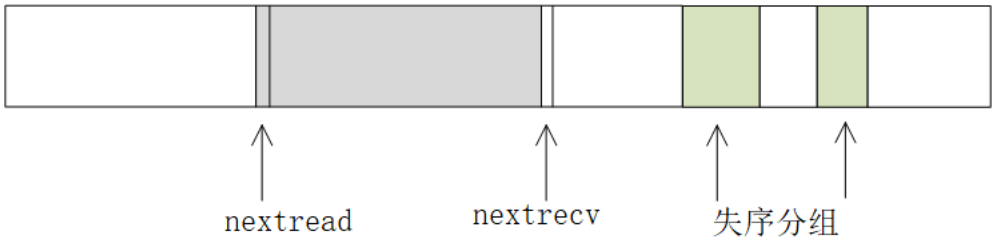

图 4: 接收缓冲区

(c) 信号量

注意到应用层的线程在缓冲区中读写数据，传输层中的线程在缓冲区中发送接收数据，存在同步互斥问题，因此使用了信号量。

3 实验步骤

3.1 协议设计

本实验使用的协议如下：

```
//传输层协议
struct packet
{
    unsigned int seq;                   //序列号
    unsigned int ack;                   //确认号
    unsigned short N;                   //滑动窗口长度
    unsigned char flag;                 //标志字段 (ACK,SYN,FIN) 0:fin 1:syn 2:ack
    unsigned short checkSum;            //检验和
    unsigned short dataLen;             //数据长度，以字节为单位
    unsigned char data[MAXBUFSIZE];     //应用层数据
};
```

各字段的含义如下：

seq 序列号

ack 确认号

N 滑动窗口大小，用于发送端通告接收端，接收端分配接收缓冲区大小

flag ACK、SYN、FIN 标志字段

checkSum 检验和字段
datalen 数据长度
data 应用层数据

3.2 发送缓冲区

建立连接后发送端需要分配发送缓冲区，编写发送缓冲区类如下

```
//发送缓冲区
class SBuf
{
public:
    SBuf(int, SOCKET, const sockaddr*);      //初始化发送缓冲区
    ~SBuf();                                 //析构发送缓冲区
    void write(const char*,int);             //应用层向发送缓冲区写数据报
    void send();                             //传输层发送数据报
    void ack();                              //接受 ack 数据报
    void retrans();                          //超时重传
    bool isterminated();                     //连接是否关闭
    bool close();                            //关闭连接
private:
    char** buf;                              //发送缓冲区
    int* len;                                //各数据报的长度
    int n;                                   //发送缓冲区的大小
    int N;                                   //滑动窗口的大小
    bool istermi;                            //连接是否关闭
    unsigned int base;                       //[base, nextSeqnum) 已发送未确认的数据报，
                                             //长度小于滑动窗口的大小 N
    unsigned int nextSeqnum;                 //[nextSeqnum, nextWrite) 应用层写入缓冲区未发送的数据报
    unsigned int nextWrite;
    bool flag;
    SOCKET s;                                //标识连接的 socket
    sockaddr addr;                           //连接对方的地址
    Timer *timer;                            //计时器
    RTO rto;                                 //计算下一次 RTO
    HANDLE mutex;                            //互斥信号量
    HANDLE slots;                            //发送缓冲区剩余大小计数信号量
    HANDLE sem_ack;                          //待确定的报文数计数信号量
};
```

其中：

1. mutex 为互斥信号量，应用层向发送缓冲区写数据、传输层发送缓冲区数据、传输层移动滑动窗口、传输层重传数据四者存在互斥
2. slots 为发送缓冲区空闲位置计数信号量，确认数据、向缓冲区中写入数据之间存在同步问题，向缓冲区中写入数据需要缓冲区有空闲位置，确认数据后缓冲区空闲位置增加，因此写入数据需要等待确认数据。
3. sem_ack 为等待确定的报文计数信号量，确认数据、发送缓冲区数据之间存在同步问题，确认数据需要缓冲区中有等待确认的分组，发送数据后，等待确认的分组数增加，因此确认数据需要等待发送数据。

3.2.1 初始化

```
/*
    n: 滑动窗口大小
    s: 连接 socket
    addr: 连接对方地址
*/
SBuf::SBuf(int n, SOCKET s, const sockaddr* addr)
{
    this->n = 2 * n;
    this->s = s;
    this->addr = *addr;
    buf = new char* [2 * n];
    len = new int[2 * n];
    memset(len, 0, 2 * n);
    memset(buf, 0, sizeof(char*) * n * 2);
    istermi = false;
    rto.DevRTT = 0;
    flag = false;
    rto.EstimatedRTT = 10;
    rto.rto = 10;
    timer = new Timer();
    N = n;
    base = nextSeqnum = nextWrite = 1;
    mutex = CreateSemaphore(NULL, 1, 1, NULL);
    slots = CreateSemaphore(NULL, 2 * n, n * 2, NULL);
    sem_ack = CreateSemaphore(NULL, 0, n, NULL);
}
```

初始化滑动窗口大小和发送缓冲区大小。初始化信号量：开始时发送缓冲区为空，slots 信号量初值和最大值为发送缓冲区大小；由于初始时未发送数据，sem_ack 信号量初值为 0，最大值为滑动窗口大小。

3.2.2 写入数据

```
/*
    将数据写入发送缓冲区
    buf: 应用层数据
    len: 数据长度
*/
void SBuf::write(const char* buf, int len)
{
    WaitForSingleObject(slots, INFINITE);
    WaitForSingleObject(mutex, INFINITE);
    this->buf[nextWrite % n] = new char[len];
    this->len[nextWrite % n] = len;
    for (int i = 0; i < len; i++)
        this->buf[nextWrite % n][i] = buf[i];
    nextWrite++;
    ReleaseSemaphore(mutex, 1, NULL);
}
```

应用层向缓冲区写入数据需要等待 slots 信号量大于 0，即发送缓冲区有空闲位置，随后将应用层数据写入缓冲区，nextWrite 递增。

3.2.3 发送分组

```
//传输层将应用层数据组装成数据报并发送
void SBuf::send()
{
    WaitForSingleObject(mutex, INFINITE);
    if (nextSeqnum < base + N && nextSeqnum < nextWrite)    //滑动窗口中还有可以发送的数据
    {
        if (base == nextSeqnum)                             //开始发送时，开启定时器
        {
            timer->setTimeOut(rto.rto);
            timer->startTimer();
            flag = true;
        }
        packet p;
        makePkt(&p, buf[nextSeqnum % n], len[nextSeqnum % n], nextSeqnum);
        sendto(s, (char*)&p, len[nextSeqnum % n] + sizeof(packet) - MAXBUFSIZE, 0, &addr,
        sizeof(sockaddr));
```

```
        nextSeqnum++;
        ReleaseSemaphore(sem_ack, 1, NULL);
    }
    ReleaseSemaphore(mutex, 1, NULL);
}
```

发送数据需要 nextseqnum < base + N 并且 nextseqnum < nextwrite，即滑动窗口未满，并且有数据要发送。如果满足条件，发送 nextseqnum 中的数据，nextseqnum 递增。发送第一个分组时开启定时器。

3.2.4 确认分组

```
//接受 ack 数据报
void SBuf::ack()
{
    sockaddr from = addr;
    int fromlen = sizeof(sockaddr);
    packet p;
    if(recvfrom(s, (char*)&p, sizeof(packet) - MAXBUFSIZE, 0, &from, &fromlen) < 0) return;
    WaitForSingleObject(mutex, INFINITE);
    if (check(&p) && isAck(p.flag) && p.ack >= base && p.ack < nextSeqnum)
    {
        if (flag)
        {
            rto.addSampleRTT(timer->getDiff());          //加入一次 RTT，估算下一次 RTO
            flag = false;
        }
        for (unsigned int i = base; i<= p.ack; i++)
        {
            delete []buf[i % n];                         //删除被确认的数据
            WaitForSingleObject(sem_ack, INFINITE);      //ack 信号量递减
        }
        int num = p.ack - base + 1;
        base = p.ack + 1;                                //base = ack + 1
        if (base == nextSeqnum)                          //发送缓冲区为空，停止计时器
            timer->stopTimer();
        else
        {
            timer->setTimeOut(rto.rto);
            timer->startTimer();                         //发送缓冲区不为空，重启定时器
        }
```

```
        ReleaseSemaphore(slots, num, NULL);
    }
    ReleaseSemaphore(mutex, 1, NULL);
}
```

如果收到检验和正确的 ack 数据报，并且 ack >= base, 则 base = ack + 1，释放 sem_ack 信号量和 slots 信号量。如果分组未经过重传被确认则加入分组的 RTT，估算下一次 RTO。

3.2.5 超时重传

```
void SBuf::retrans()
{
    WaitForSingleObject(mutex, INFINITE);
    if (timer->isStart && timer->testTimeOut())
    {
        flag = false;
        timer->startTimer();
        //定时器超时，重传 [base, nextSeqnum) 中的数据
        for (unsigned int i = base; i < nextSeqnum; i++)
        {
            packet p;
            makePkt(&p, buf[i % n], len[i % n], i);
            sendto(s, (char*)&p, len[i % n] + sizeof(packet) - MAXBUFSIZE, 0, &addr,
            sizeof(sockaddr));
        }
    }
    ReleaseSemaphore(mutex, 1, NULL);
}
```

若定时器超时，重传 [base, nextseqnum) 中的分组。

3.3 接收缓冲区

接收端在收到发送端建立连接请求之后会根据接收端滑动窗口的大小分配接收缓冲区，用于接收发送端发送的分组，并向应用层交付数据。编写接收缓冲区类如下:

```
//接收缓冲区
class RBuf
{
public:
    RBuf(int, SOCKET, SOCKET, sockaddr);     //初始化接收缓冲区
```

```
    ~RBuf();
    void read(char*, int);                  //应用层读接受缓冲区的数据
    bool recv();                            //rdt 接收数据，放入缓冲区
    void getSocket(SOCKET*);                //返回连接 socket
private:
    char** buf;                             //缓冲区
    int n;                                  //缓冲区大小
    unsigned int front;                     //缓冲区队首，指向应用层下一个要读的数据报
    unsigned int rear;                      //缓冲区队尾，指向按序接收到的最后一个数据报，之后是失序数据报
    unsigned int expected;                  //接收缓冲期待接收的下一数据报的序列号
    SOCKET cs;                              //连接 socket
    SOCKET s;                               //服务器 socket
    sockaddr addr;                          //对方的地址
    HANDLE mutex;                           //互斥信号量
    HANDLE slots;                           //缓冲区剩余位置计数信号量
    HANDLE items;                           //缓冲区按序接收到的数据报数目计数信号量
};
```

1. mutex 为互斥信号量，应用层从缓冲区读数据、接收缓冲区接收数据存在互斥
2. slots 为接收缓冲区空余位置计数信号量，接收缓冲区接收数据、应用层读取数据存在同步问题，接收数据需要接收缓冲区有空余位置，应用层读取数据后接收缓冲区空余位置增加，因此接收数据需要等待应用层读取数据
3. items 为接收缓冲区中按序到达的分组计数信号量，应用层读取数据需要接收缓冲区中存在按序到达的分组，因此该信号量作为应用层可以读取数据的标志。

3.3.1 初始化

```
/*
    n: 接收缓冲区大小
    cs: 连接 socket
    s: 服务器 socket
    addr: 连接对方地址
*/
RBuf::RBuf(int n, SOCKET cs, SOCKET s,sockaddr addr)
{
    this->n = n;
    this->s = s;
    this->cs = cs;
    this->addr = addr;
    buf = new char* [n];
```

```
14
15
16      memset(buf, 0, sizeof(char*) * n);
17      front = rear = 1;
18      expected = 1;
19
20      mutex = CreateSemaphore(NULL, 1, 1, NULL);
21      slots = CreateSemaphore(NULL, n, n, NULL);
22      items = CreateSemaphore(NULL, 0, n, NULL);
23  }
```

初始化接收缓冲区的大小，创建信号量。

3.3.2 读取数据

```
1       /*
2       buf: 应用层接收的数据报
3       len: 数据报长度
4   */
5   void RBuf::read(char* buf, int len)
6   {
7       WaitForSingleObject(items, INFINITE);
8       WaitForSingleObject(mutex, INFINITE);
9       front++;
10      for (int i = 0; i < len; i++)
11          buf[i] = this->buf[front % n][i];
12      delete []this->buf[front % n];
13      this->buf[front % n] = NULL;
14      ReleaseSemaphore(mutex, 1, NULL);
15      ReleaseSemaphore(slots, 1, NULL);
16  }
```

当 items 信号量大于 0，即接收缓冲区中存在按序到达的分组时，向应用层传递数据，否则发生阻塞。传递数据后，删除分组，接收缓冲区空余位置增加，slots 信号量递增。

3.3.3 接收数据

```
1   bool RBuf::recv()
2   {
3       /******/
```

```
    if(recvfrom(s, (char*)&p, sizeof(packet), 0, &from, &fromlen) < 0) return true;
    WaitForSingleObject(mutex, INFINITE);
    if (check(&p) && p.seq >= expected && !buf[(rear + p.seq - expected + 1) % n]
    && !isSyn(p.flag))    //数据报检验和正确
    {
        packet ackPkt;
        WaitForSingleObject(slots, INFINITE);
        int idx = rear + p.seq - expected + 1;    //接收的数据报在缓存中的位置
        buf[idx % n] = new char[p.dataLen];
        for (int i = 0; i < p.dataLen; i++)        //将按序或失序的数据报缓存起来
            buf[idx % n][i] = p.data[i];
        while (buf[(rear + 1)% n])                //得到当前按序接收的最后一个数据报的序号
        {
            ReleaseSemaphore(items, 1, NULL);    //计数信号量递增
            rear++;
            expected++;
        }
        makeAckPkt(&ackPkt, expected - 1);
        sendto(s, (char*)&ackPkt, sizeof(packet) - MAXBUFSIZE, 0, &addr,

        sizeof(sockaddr));    //发送累计确认 ack
    }
    else    //数据报检验和错误或重复接收
    {
        packet ackPkt;
        makeAckPkt(&ackPkt, expected - 1);
        sendto(s, (char*)&ackPkt, sizeof(packet) - MAXBUFSIZE, 0, &addr, sizeof(sockaddr));
    }
    ReleaseSemaphore(mutex, 1, NULL);
    return flag;
}
```

当数据检验和正确并且未缓存时，需要等待 slots 信号量大于 0，即接收缓冲区有空余位置后，将数据缓存到接收缓冲区，向发送端发送累计确认 ack，items 信号量根据新增加的按序分组数递增。

4 实验结果

设置滑动窗口大小为 320，4 个测试文件的传输时间和平均吞吐率如表1所示。滑动窗口大小对传输的影响将在任务 3-4 中详细进行测试。

文件名称	文件大小 (byte)	传输时间 (ms)	平均吞吐率
1.jpg	1857353	843.81	17196.5
2.jpg	5898505	4039.53	11407.8
3.jpg	11968994	7110.09	13151.4
helloworld.txt	1655808	1874.86	6899.72

表 1: 测试文件传输结果

图 5: 发送端

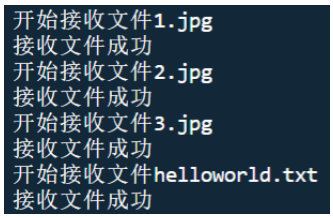

图 6: 接收端