

计算机网络大作业第三部分

周延霖 信息安全 2013921

实验要求

在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

实验设计

实验原理

1. 网络拥塞

主机发送的数据过多或过快，造成网络中的路由器(或其他设备)无法及时处理，从而引入时延或丢弃。

2. 拥塞控制

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络性能就要变坏，这种情况就叫做网络拥塞。在计算机网络中数位链路容量（即带宽）、交换结点中的缓存和处理机等，都是网络的资源。若出现拥塞而不进行控制，整个网络的吞吐量将随输入负荷的增大而下降。

目标：既不造成网络严重拥塞，又能更快地传输数据。

带宽探测：接收到ACK，提高传输速率；发生丢失事件，降低传输速率。

ACK返回：说明网络并未拥塞，可以继续提高发送速率。

丢失事件：假设所有丢失是由于拥塞造成的，降低发送速率。

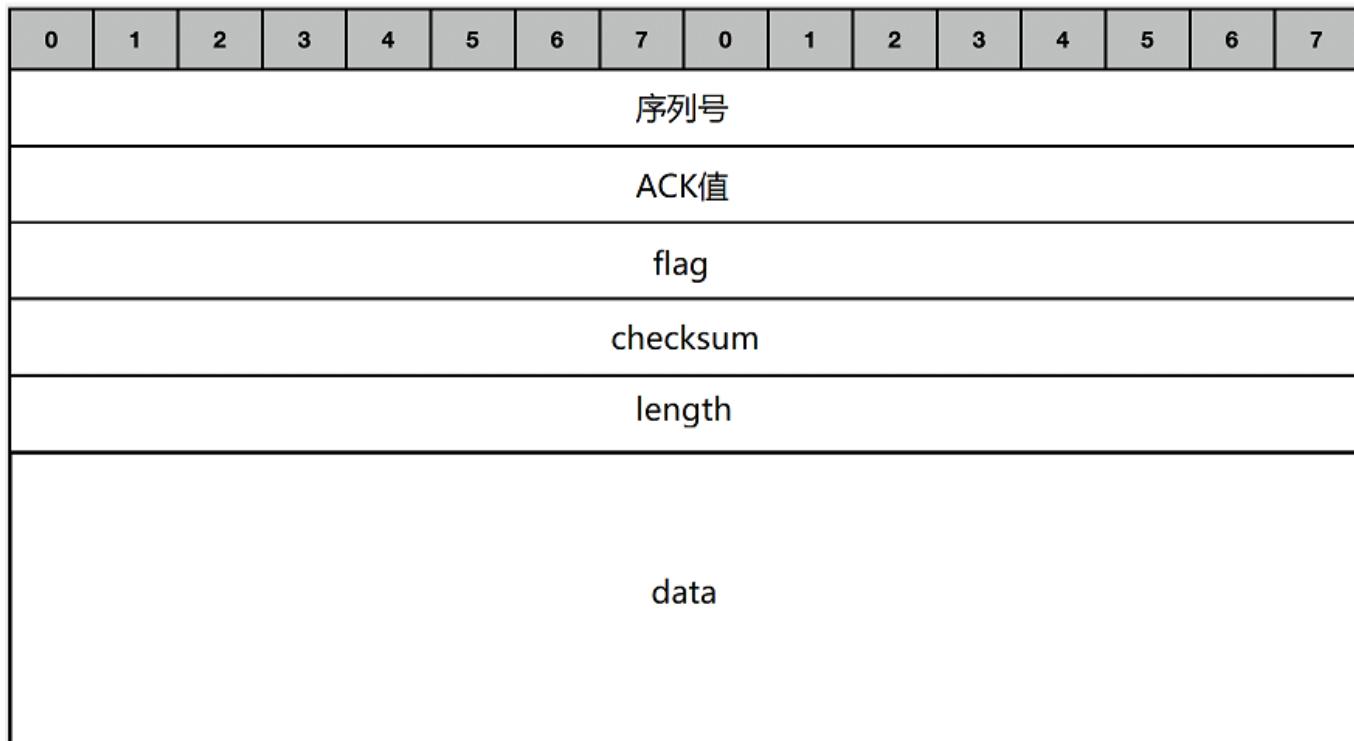
3. 拥塞窗口

发送方维持一个拥塞窗口cwnd的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。

发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

协议设计

1. 报文格式

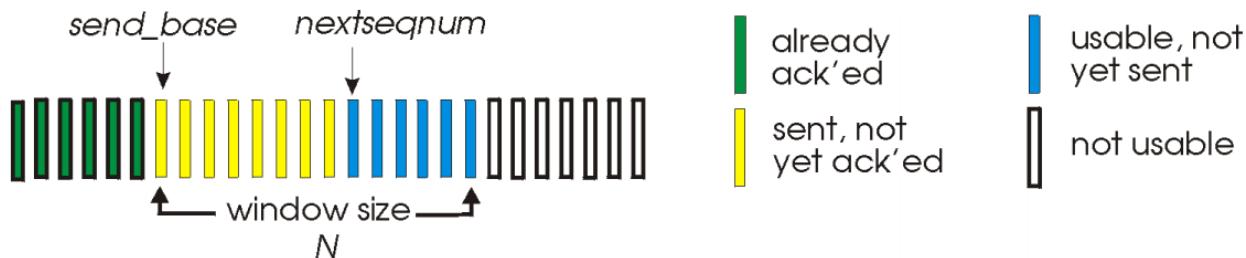


报文由10字节的定长报文首部和变长的数据部分组成。

首部各字段作用如下：

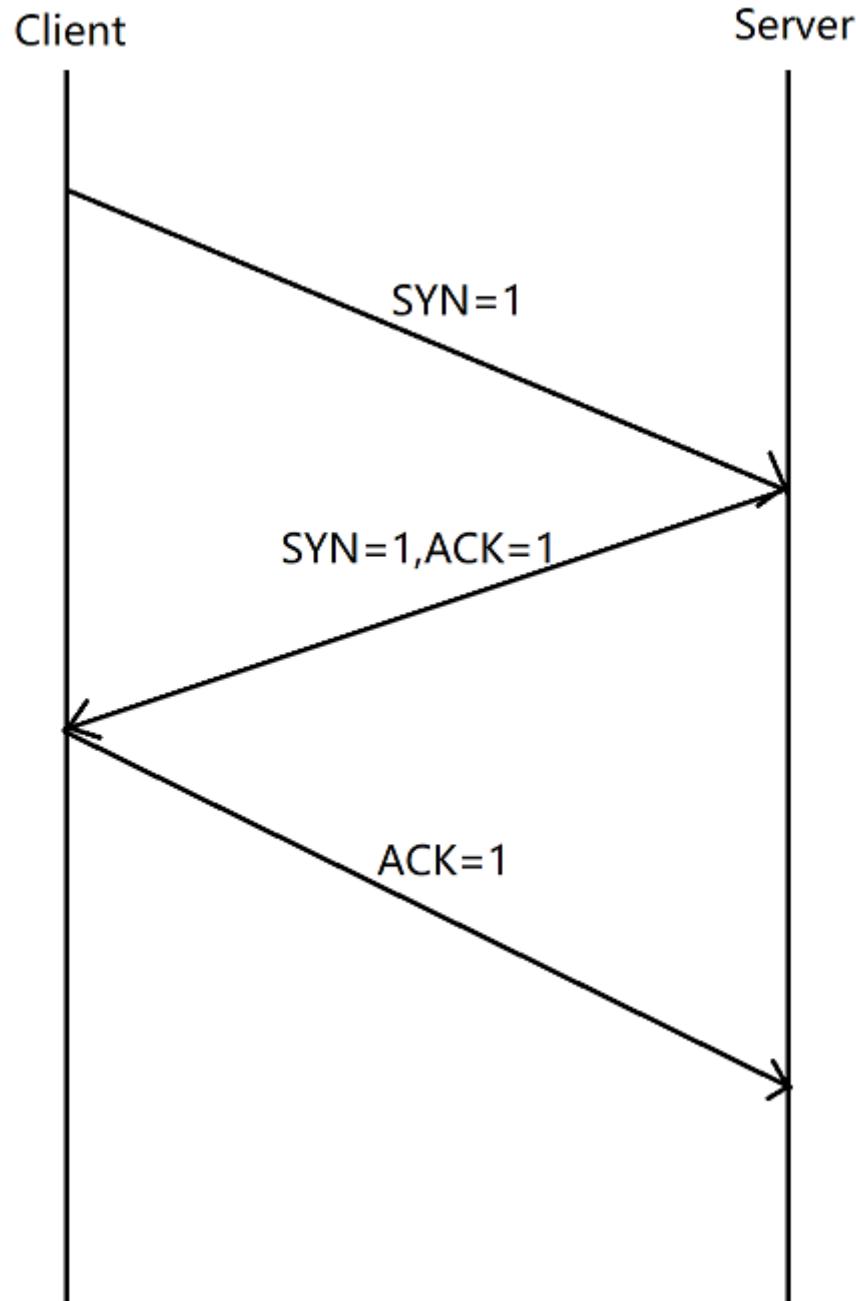
- 0-15位为传输的数据报的序列号，序列号空间为0-65535。
- 16-31位为ack确认值，用于服务器对接收到的客户端的数据进行确认。
- 32-47位为标志位，目前只使用低5位，从低至高分别为SYN、ACK、FIN、LAS、RST，其中LAS位用于标识当前数据包是承载着所发送文件的最后一个数据包。
- 48-63位为校验和，用于实现差错检测。
- 64-79位为数据部分的长度。

2.滑动窗口



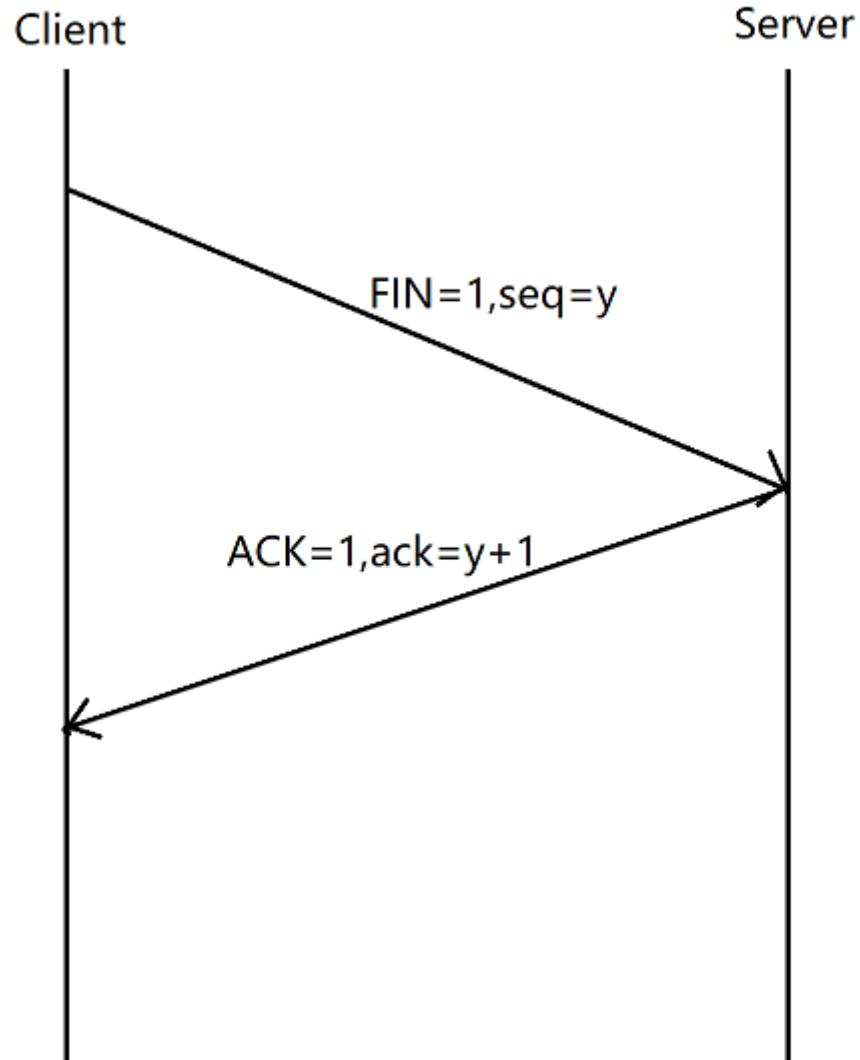
如上图所示，窗口分为左边界`send_base`、发送边界`nextseqnum`和右边界，窗口大小固定。窗口左边界左侧为已经发送并得到确认的数据，左边界到发送边界的为已发送但未得到确认的数据，发送边界到右边界为等待发送的数据，右边界右侧为不可发送的数据。发送数据时，判断`nextseqnum`是否超过右边界，如果没有超过则发送序列号为`nextseqnum`的数据包。

3.握手与挥手



- 三次握手进行连接
 - 这里我使用类似于TCP的三次握手进行连接建立。首先，客户端向服务端发送请求数据包，将SYN置位。服务端接收到数据包后，向客户端发送确认数据包，数据包中SYN和ACK均置位。客户端接收到数据包后，向服务端发送数据包，将ACK置位，之后进入连接建立状态，开始向服务器发送数据。服务端接收到ACK数据包后，进入连接建立状态，开始进行数据的接收。
- 握手过程中丢包的处理
 1. 如果客户端发送的第一个SYN数据包产生了丢失，客户端迟迟无法接收到来自服务端的SYN、ACK数据包，则客户端会对SYN数据包进行超时重传，当超过重传次数后若还未收到响应，则建立连接失败。
 2. 如果服务端接收到了客户端的SYN数据包，但其发回客户端的SYN、ACK数据包产生了丢失。此时，客户端仍无法接收到SYN、ACK数据包，客户端会继续重传SYN报文；而服务端则迟迟无法接收到第三次握手来自客户端的ACK数据包，此时服务器也会对SYN、ACK数据包进行超时重传，同样，当超过重传次数后若还未收到响应，则建立连接失败。
 3. 如果客户端发送给服务器的ACK数据包产生了丢失。此时，客户端已经接收到了来自服务器的SYN、ACK报文，并向服务器发送ACK报文后，客户端进入了建连状态，可以开始向服务器发送

具体的数据。而如果服务器在等待第三次握手的ACK报文时接收到了来自客户端发送的承载着具体数据的报文，服务器会认为产生了异常的连接，其会向客户端发送一个RST数据包，表示连接异常，建立连接失败。

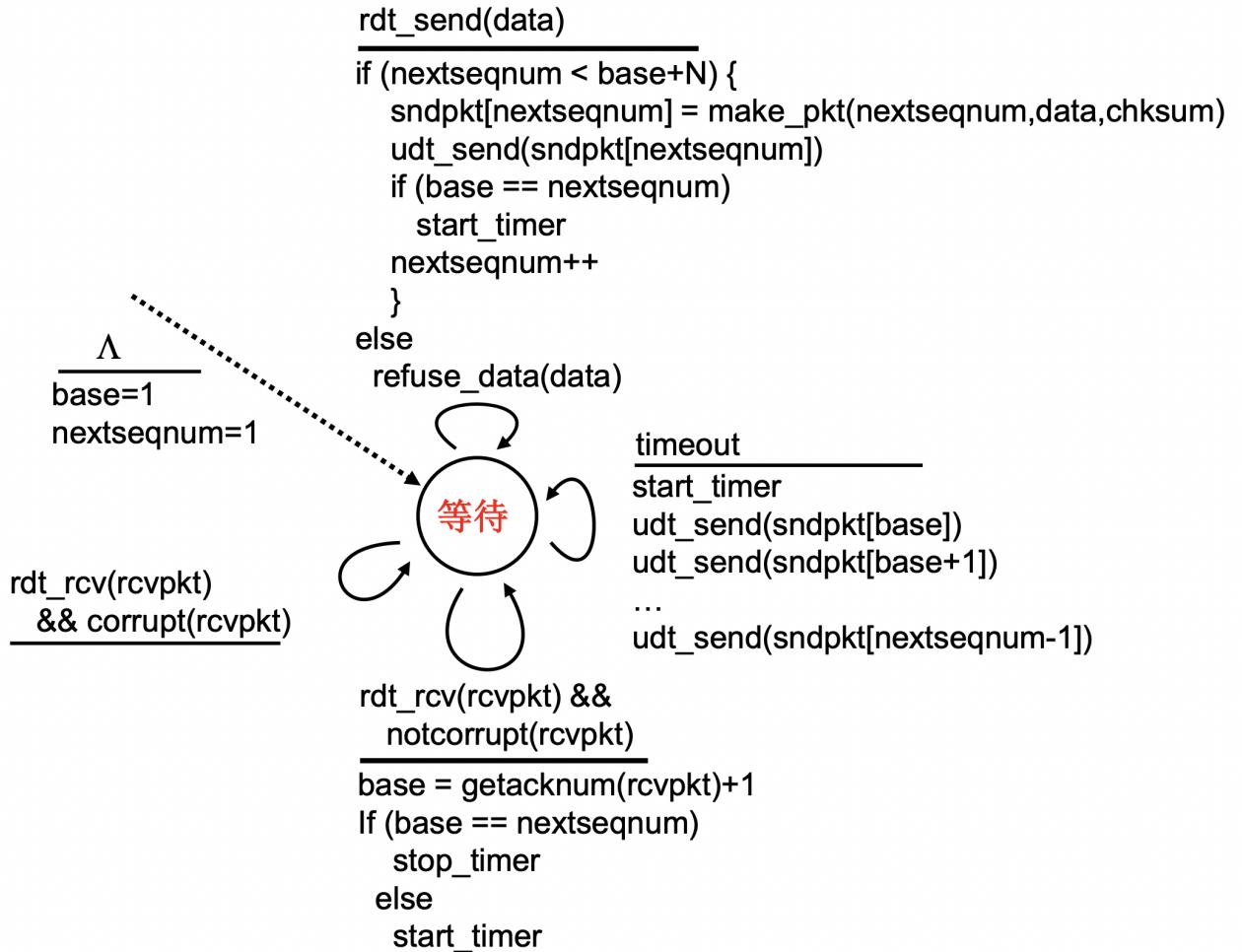


- 两次挥手断开连接
 - 挥手方面，由于在实验中，发送端和接收端是固定的，服务器不会主动向客户端发送数据，我们只需要进行两次挥手即可。首先，客户端向服务器发送数据包，将FIN置位，并将seq填充一个随机值y，由于本次序列号只用了0和1，这里是为了避免对FIN报文进行确认的ACK数据包和对普通数据报文进行确认的ACK数据包混淆。服务端接收到报文后，向客户端发送响应报文，将ACK置位，并将ack值设置为y+1，然后断开连接。客户端接收到数据包后，连接成功断开。
- 挥手过程中丢包的处理
 1. 如果客户端向服务器发送的FIN报文产生了丢失，客户端将迟迟无法接收到来自服务器的ACK报文，其会对FIN报文进行超时重传，重传超过一定次数之后会默认此时连接产生了异常，会直接断开连接。对于服务器来说，服务器接收到LAS标志位为1的报文而进入挥手状态后，会开启一个计时器，设置一个超时时间，如果超过该时间仍未接收到来自客户端的FIN报文，服务器也会默认连接产生了异常，会直接断开连接。
 2. 如果服务器向客户端发送的ACK确认报文产生了丢失，此时服务器已经接收到了来自客户端的FIN报文，向客户端发送ACK报文后服务器便会断开连接。此时，客户端由于没有接收到ACK报文，会继续重传FIN报文，直到超过最大重传次数。

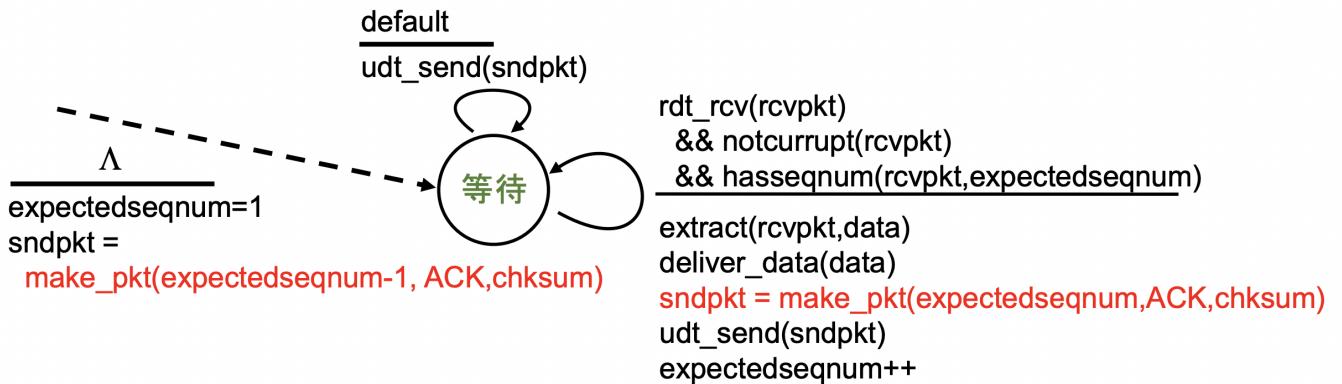
4. 数据传输

对于数据的发送和接收采用课本中GBN的状态机。

发送端状态机：



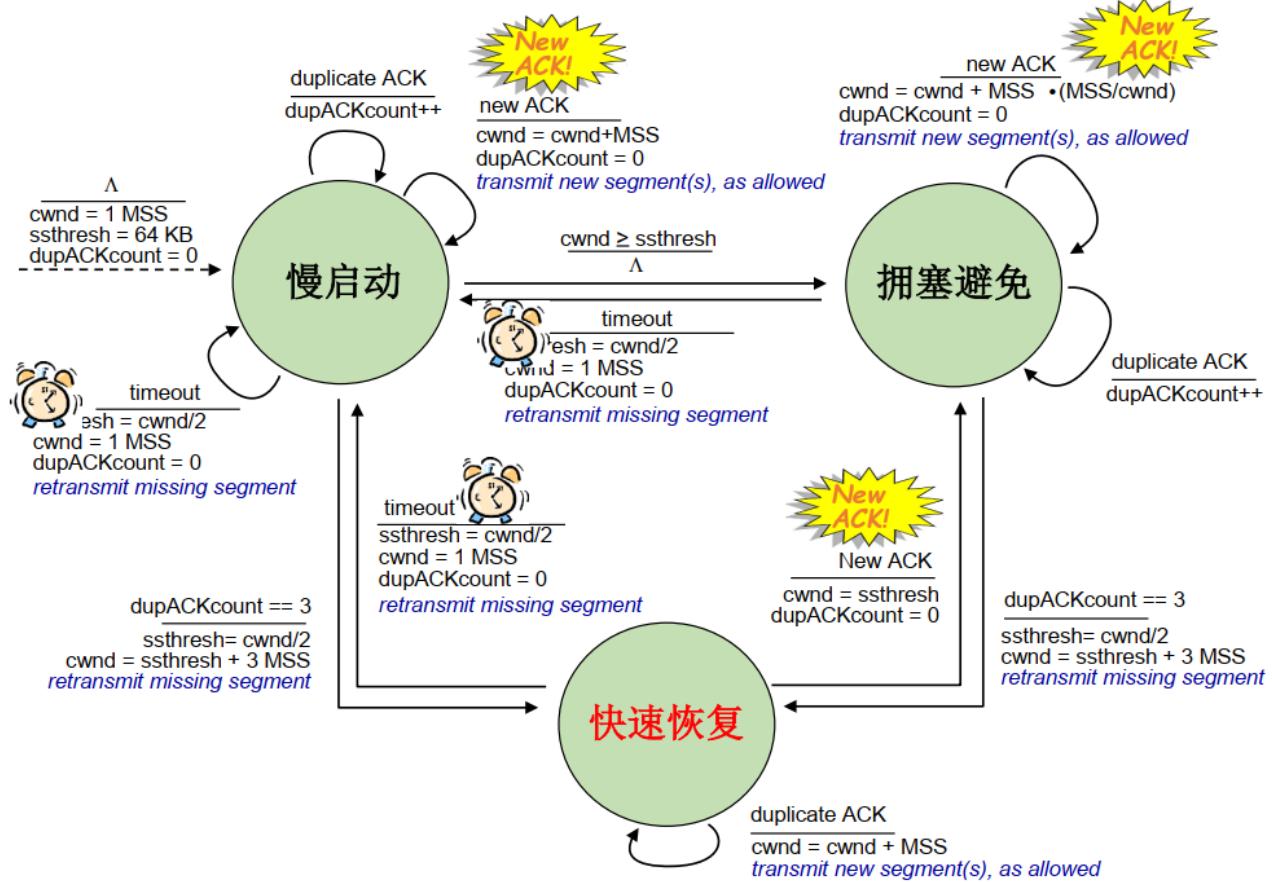
接收端状态机：



数据在传输时，将一个文件封装到多个报文中进行分段传输。在传输时，发送端无需等待接收到上一个发送包的序号的ACK报文便能发送下一个数据包，可以继续发送直到已发送但未确认的数据包的数目达到窗口大小。接收端接收到一个数据包，先进行差错校验，如果检查无误，而且该报文的序列号为接收端的期望的序列号，则会接受该报文，并向发送端发回一个对该报文序列号的ACK报文；如果该报文的序列号不是接收端期望的序列号，接收端会将该报文丢弃，并向发送端发回一个对最后一次正确收到的序列号的ACK报文。发送端如果长时间没有接收到任何对已发送的数据包的ACK报文，则其会进行超时重传，将当前窗口中所有的已发送但未确认的报文重发一次。

在最后，发送方需要向接收端发送一个LAS=1的报文，以通知接收端文件传输结束，进入挥手阶段。

5. 拥塞控制reno算法



三个阶段：

- 慢启动阶段：初始拥塞窗口 $cwnd=1$ ，每收到一个新的ACK， $cwnd$ 增1。当连接初始建立或超时时，进入慢启动阶段。
- 拥塞避免阶段：设置阈值 $ssthresh$ ，当拥塞窗口 $cwnd>=ssthresh$ 时，从慢启动阶段进入拥塞避免阶段。拥塞避免阶段中，每轮rtt， $cwnd$ 增1。
- 快速恢复阶段：当接收到三次冗余ACK时，进入快速恢复阶段。快速恢复阶段中，每收到一个冗余ACK， $cwnd$ 增1。如果出现超时，进入慢启动阶段；如果接收到新的ACK，进入拥塞避免阶段。

主要事件：

- 超时：当超时发生时， $ssthresh$ 设置为 $cwnd/2$ ， $cwnd$ 设置为1，并进入慢启动阶段。
- 接收到新的ACK：如果当前处于快速恢复阶段，接收到新的ACK后会进入拥塞避免阶段， $cwnd=ssthresh$ 。如果当前处于其余两个阶段，则会对 $cwnd$ 进行相应的增加。
- 三次冗余ACK：接收到三次冗余ACK， $ssthresh$ 设置为 $cwnd/2$ ， $cwnd$ 设置为 $ssthresh+3$ ，并进入快速恢复阶段。

代码实现

这里只展示本次实验相对于第二次实验代码的大的修改部分。由于拥塞控制主要由发送端进行，本次实验中并未对接收端代码进行改动。

拥塞控制类

为了实现拥塞控制reno算法，我实现了一个congestionController拥塞控制类，具体代码如下。

```
class congestionController {  
private:  
    u_int cwnd; // 拥塞窗口  
    u_int ssthresh; // 阈值  
    u_int dupACKcount; // 冗余ACK数目  
    u_int step; // 进入拥塞避免阶段后，cwnd增长所需的步长，接收到step个NEW ACK后，  
    cwnd增1  
    enum {  
        SLOWSTART,  
        CONGESTIONAVOID,  
        QUICKRECOVER  
    } state;  
public:  
    congestionController() { // 初始时，处于慢启动阶段，cwnd=1, ssthresh这里  
    我设置为8  
        cwnd = 1;  
        ssthresh = INITIALSSTHRESH; // INITIALSSTHRESH是个宏，8  
        dupACKcount = 0;  
        state = SLOWSTART;  
    };  
    u_int getCWND() {  
        return cwnd;  
    }  
    u_int getDupACK() {  
        return dupACKcount;  
    }  
    void newACK() { // 接收到一个新的ACK  
        dupACKcount = 0;  
        switch(state) {  
            case SLOWSTART: {  
                cwnd += 1;  
                if (cwnd >= ssthresh) {  
                    step = cwnd;  
                    state = CONGESTIONAVOID;  
                }  
            }break;  
            case CONGESTIONAVOID: {  
                step -= 1;  
                if (step <= 0) {  
                    cwnd += 1;  
                    step = cwnd;  
                }  
            }break;  
            case QUICKRECOVER: {  
                cwnd = ssthresh;  
                state = CONGESTIONAVOID;  
                step = cwnd;  
            }break;  
        }  
    }  
}
```

```

        }
    }

    void timeOut() { // 超时发生
        dupACKcount = 0;
        ssthresh = cwnd / 2;
        cwnd = 1;
        state = SLOWSTART;
    }

    void dupACK() { // 接收到一个冗余ACK
        dupACKcount++;
        if (dupACKcount < 3) {
            return;
        }
        switch(state) {
        case SLOWSTART:
        case CONGESTIONAVOID: {
            ssthresh = cwnd / 2;
            cwnd = ssthresh + 3;
            state = QUICKRECOVER;
        }break;
        case QUICKRECOVER: {
            cwnd += 1;
        }break;
        }
    }
};

}

```

发送端数据发送函数

发送数据时，我们既需要使已发送的窗口大小小于发送窗口大小，也需要使其小于拥塞窗口大小。

```

void rdt_send(SOCKET *server, SOCKADDR_IN *server_addr, char *msg, int
len, bool last = false) {
    assert(len <= MSS);
    // 窗口满了就阻塞
    while ((u_short)(nextseqnum - base) >= WND || (u_short)(nextseqnum -
base) >= my_cong_controller.getCWND()) { // 还需要判断是否小于拥塞窗口大小
        continue;
    }
    Packet *packet = new Packet;
    packet->hdr.set_args(nextseqnum, 0, last ? LAS : 0, 0, len);
    memcpy(packet->data, msg, len);
    u_short checksum = checksum((char*)packet, sizeof(Header) + len);
    packet->hdr.checksum = checksum;
    buffer_lock.lock();
    gbn_buffer.push_back(packet);
    buffer_lock.unlock();
    sendto(*server, (char*)packet, len + sizeof(Header), 0,
(sockaddr*)server_addr, sizeof(SOCKADDR_IN));
    print_lock.lock();
    cout << "向服务器发送数据包，首部为: seq:" << packet->hdr.seq << ", ack:"
}

```

```

<< packet->hdr.ack << ", flag:" << packet->hdr.flag << ", checksum:" <<
packet->hdr.checksum << ", len:" << packet->hdr.length << ", 剩余发送窗口大
小:" << WND - (nextseqnum - base) - 1 << endl;
    print_lock.unlock();
    if (base == nextseqnum) {
        my_timer.start_timer();
    }
    nextseqnum += 1;
}

```

发送端接收线程函数

修改接收线程函数，在相应事件发生时调用拥塞控制类的函数，实现拥塞窗口动态变化。

```

void receive_thread(SOCKET *server, SOCKADDR_IN *server_addr) {
    // 开启非阻塞模式
    u_long mode = 1;
    ioctlsocket(*server, FIONBIO, &mode);
    char *recv_buffer = new char[sizeof(Header)];
    Header *header;
    SOCKADDR_IN ser_addr;
    int ser_addr_len = sizeof(SOCKADDR_IN);
    while (true) {
        if (send_over) {
            mode = 0;
            ioctlsocket(*server, FIONBIO, &mode);
            delete []recv_buffer;
            return;
        }
        while (recvfrom(*server, recv_buffer, sizeof(Header), 0,
(sockaddr*)&ser_addr, &ser_addr_len) <= -1) {
            if (send_over) {
                mode = 0;
                ioctlsocket(*server, FIONBIO, &mode);
                delete []recv_buffer;
                return;
            }
            if (my_timer.time_out()) {
                my_cong_controller.timeOut(); // 发生了超时事件
                for (auto packet : gbn_buffer) {
                    sendto(*server, (char*)packet, sizeof(Header) +
packet->hdr.length, 0, (sockaddr*)server_addr, sizeof(SOCKADDR_IN));
                    print_lock.lock();
                    cout << "超时重传数据包, 首部为: seq:" << packet->hdr.seq
<< ", ack:" << packet->hdr.ack << ", flag:" << packet->hdr.flag << ",
checksum:" << packet->hdr.checksum << ", len:" << packet->hdr.length <<
endl;
                    print_lock.unlock();
                }
                my_timer.start_timer();
            }
        }
    }
}

```

```

    }
    header = (Header*)recv_buffer;
    int cksum = checksum(recv_buffer, sizeof(Header));
    if (cksum != 0) {
        continue;
    }
    if (header->flag == RST) {
        cout << "连接异常, 退出程序" << endl;
        exit(-1);
    }
    else if (header->flag == ACK) {
        if (header->ack >= base) {
            my_cong_controller.newACK(); // 接收到新的ACK
        }
        else {
            my_cong_controller.dupACK(); // 接收到重复ACK
            if (my_cong_controller.getDupACK() == 3) { // 如果收到了3个重
                复ACK, 这里会快速重传base处的数据包
                    Packet *packet = gbn_buffer[0];
                    sendto(*server, (char*)packet, sizeof(Header) +
packet->hdr.length, 0, (sockaddr*)server_addr, sizeof(SOCKADDR_IN));
                    print_lock.lock();
                    cout << "接收到三个重复ACK, 快速重传数据包, 首部为: seq:" <<
packet->hdr.seq << ", ack:" << packet->hdr.ack << ", flag:" << packet-
>hdr.flag << ", checksum:" << packet->hdr.checksum << ", len:" << packet-
>hdr.length << endl;
                    print_lock.unlock();
                }
            }
            int recv_num = header->ack + 1 - base;
            for (int i = 0; i < recv_num; i++) {
                buffer_lock.lock();
                if (gbn_buffer.size() <= 0) {
                    break;
                }
                delete gbn_buffer[0];
                gbn_buffer.erase(gbn_buffer.begin());
                buffer_lock.unlock();
            }
            base = header->ack + 1;
            print_lock.lock();
            cout << "接收到来自服务器的数据包, 首部为: seq:" << header->seq <<
", ack:" << header->ack << ", flag:" << header->flag << ", checksum:" <<
header->checksum << ", len:" << header->length << ", 拥塞窗口大小:" <<
my_cong_controller.getCWND() << endl;
            print_lock.unlock();
        }
        if (base != nextseqnum) {
            my_timer.start_timer();
        }
        else {
            my_timer.stop_timer();
        }
    }
}

```

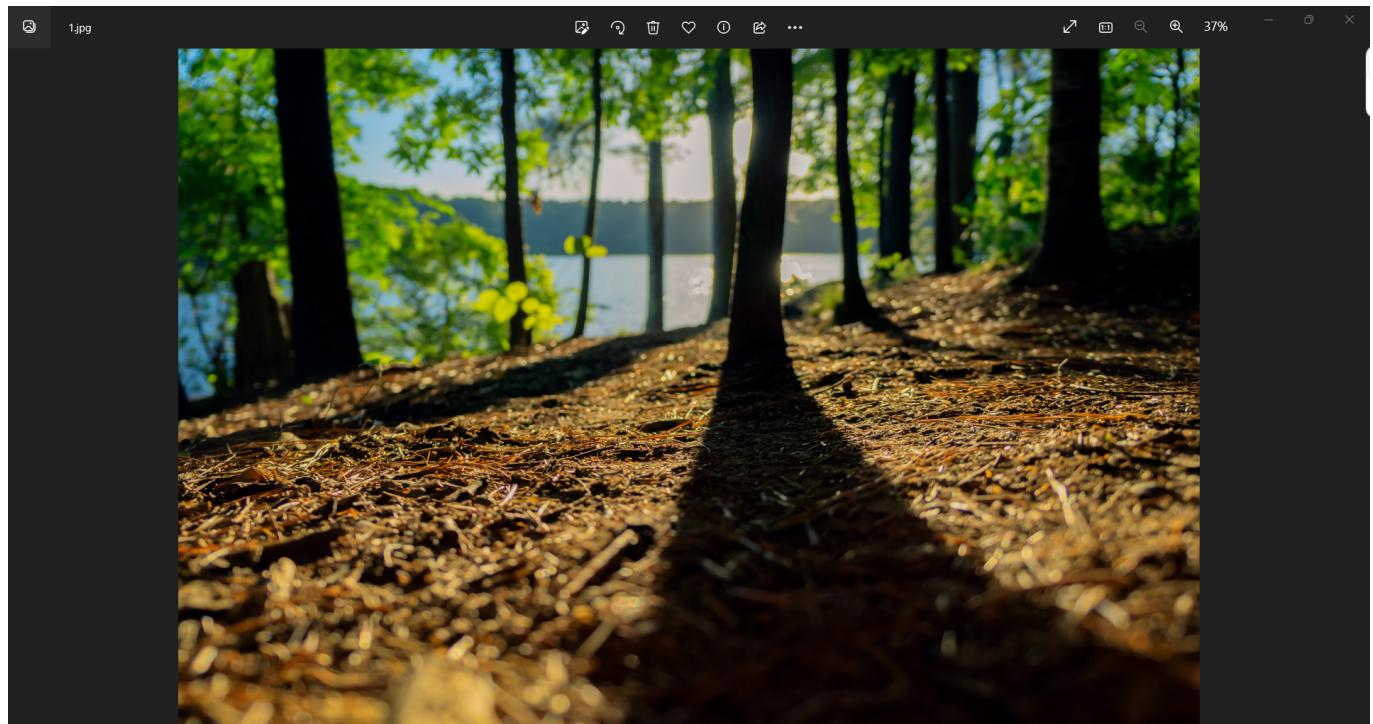
```
}
```

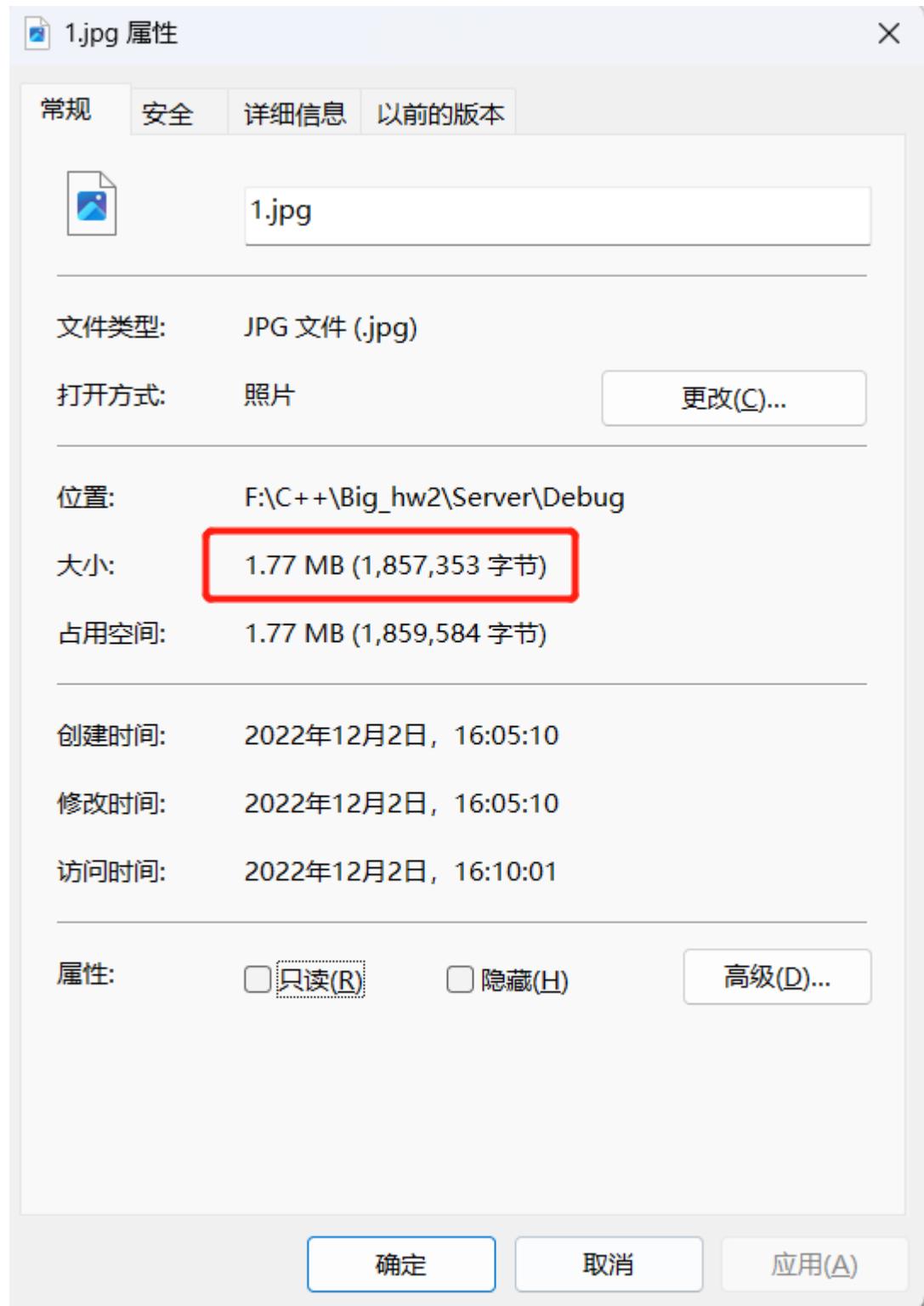
实验结果展示

握手与挥手和实验前两部分一致，在此略过。

数据传输

传输结果





如上图所示，传输结果与原图大小、信息完全相同，传输成功。