天津大學

《计算机网络实践》课程报告

TCP 的设计与实现

学　号

姓　名

学　院

专　业

年　级

任课教师

日期　2021年8 月20 日

一、报告摘要

本次实践需要实现TCP协议的连接管理，可靠传输，流量控制及拥塞控制功能：设计为通过三次握手和四次挥手来实现建立连接和拆除连接；通过设计发送接收缓冲区及滑动窗口来实现TCP的累计确认，超时重传，快速重传等机制；通过TCP包头`advertised_window`字段及`TEST`检测包来实现流量控制机制；使用慢启动、拥塞避免、快速恢复等算法实现拥塞控制机制。实践很好的实现了TCP协议的这些功能。

二、 任务分析

1、连接管理

- TCP三次握手
  - 服务端进入listen状态后，从收到客户端发来的SYN报文中拿到对端的IP和PORT；
  - LISTEN状态的socket 维护两个列表；
  - 完成三次握手后socket状态进入establish；
  - 将建立了连接的socket放入内核已建立连接哈希表中。
- TCP四次挥手
  - Peer 两端，哪一端会进入 TIME_WAIT；
  - 如何处理TCP 的 Peer 两端同时断开连接。

2、流量控制：

- 如何进行流量控制；
- 当发送方停止发送数据后，如何才能知道自己可以继续发送数据。

3、可靠传输

- 发送数据
  - 构造数据包，数据包长度；
  - 什么时候发送数据包，发送多少数据包。
- 接收数据
  - 要不要发 要怎么发ACK；
  - 如何整理接收到的TCP包；
  - 如何把接收到的一个个分开的包含头部的TCP包去掉header 将可靠连续的数据写入接收缓冲区 等待用户读取。
- 丢包/错误重传
  - 如何判定丢包/错误，什么时候需要重传；
  - 选择什么重传机制；
  - 设置合适的timer。
- 如何解决乱序问题

4、拥塞避免

- 判断当前网络是否进入拥塞导致丢包；
- 网络拥塞后如何调整窗口大小避免丢包。

三、 协议设计

（一）总体设计

1、连接管理

- 完成TCP三次握手，使socket进入establish状态；
- 完成TCP四次挥手，使socket进入closed状态。

2、流量控制

- 接收方在接收到数据后在返回给发送方的信息中携带下一次传输的数据量。

3、可靠传输

- 写入和读取发送缓冲区和接受缓冲区；
- 检测错误和丢包的发生；
- 解决错误和丢包问题；
- 解决乱序问题。

4、拥塞控制

- 使用慢启动、拥塞避免、快速重传、快速恢复算法进行拥塞控制。

（二）数据结构设计

1、协议头部数据结构

```
typedef struct {
    uint16_t source_port;       //2 bytes 源端口
    uint16_t destination_port;  //2 bytes 目的端口
    uint32_t seq_num;           //4 bytes sequence number
    uint32_t ack_num;           //4 bytes ack number
    uint16_t hlen;              //2 bytes 包头长 这个项目里全是20
    uint16_t plen;              //2 bytes 包总长 包括包头和包携带的数据 20+数据长度
    uint8_t flags;              //1 byte  标志位 比如 SYN FIN ACK 等
    uint16_t advertised_window; //2 bytes 接收方发送给发送方的建议窗口大小 用于流量控制
    uint8_t ext;                //1 byte  一些额外的数据
} tju_header_t;
```

2、TCP报文数据结构

```
typedef struct {
    tju_header_t header;
    struct timeval sent_time;
    char* data;
} tju_packet_t;
```

3、发送缓冲区结构

```
typedef struct skb_node
{
    char *data;
    int len;
    int seq;
    int flag;
    struct skb_node* next;
```

```
    struct skb_node* prev;
    struct timer* skb_timer; //计时器
    struct timeval send_time;//记录发送时间，用于计算rto
}skb_node;//缓冲区节点

typedef struct send_queue
{
    skb_node* head;
    skb_node* tail;
}send_queue;//发送缓冲区
```

4、接收缓冲区结构

```
typedef struct recv_skb_node
{
    char *data;
    int len;
    int seq;
    int flag;
    struct recv_skb_node* next;
    struct recv_skb_node* prev;
}recv_skb_node;//缓冲区节点

typedef struct recv_queue
{
    recv_skb_node* head;
    recv_skb_node* tail;
}recv_queue;//发送缓冲区
```

5、发送窗口结构

```
typedef struct {
    uint16_t window_size;
    uint16_t rwnd;
    uint16_t cwnd;
    uint32_t nextseq;
    uint32_t base;//未确认数据包
    int wait_for_ack;//等待确认的数据大小
    timer* persist_timer;
    int ack_cnt;//重复ACK数
    uint32_t estmated_rtt;
    uint32_t rtt_var;
    uint32_t timeout;//超时时长
} sender_window_t;
```

6、接收窗口结构

```
typedef struct {
    uint32_t expect_seq;
    recv_queue* window;
    int recv_len;
} receiver_window_t;
```

7、计时器结构

```
typedef struct timer
{
    int now_time;
    int set_time;
    void (*func)(tju_tcp_t* sock);//超时后执行的函数
}timer;
```

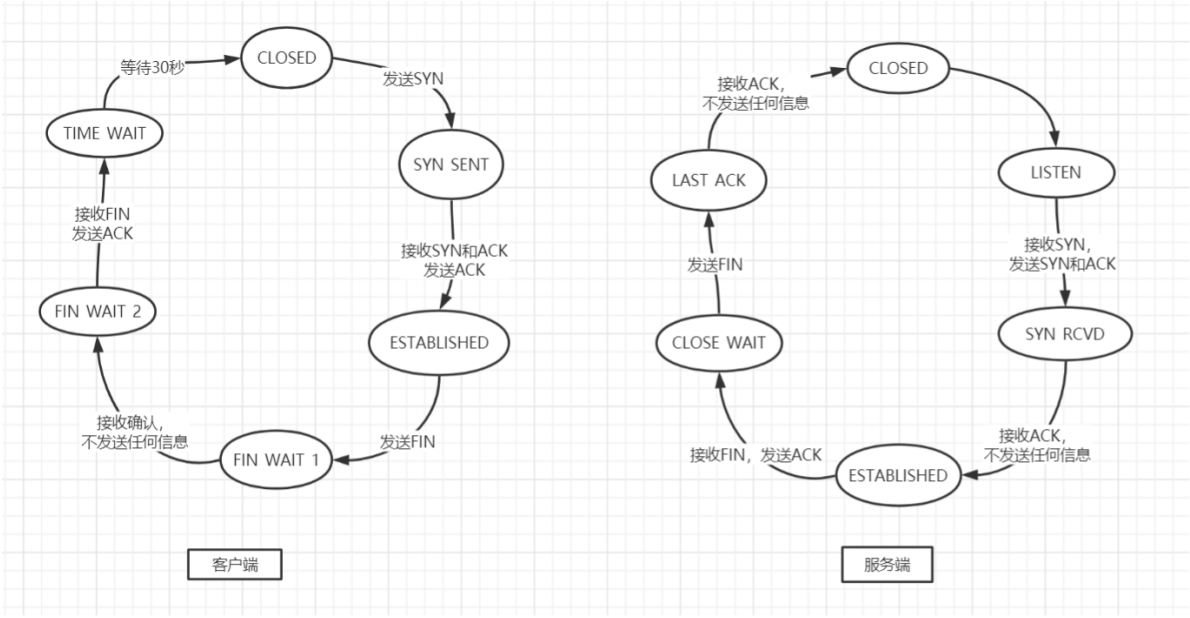
（三）协议规则设计

1、TCP三次握手建立连接

- 第一次握手：主动连接上服务器, 发送SYN ，这时客户端的状态是SYN_SENT，服务器等待客户的连接，收到客户端的 SYN 后，若半连接队列未满，服务器将该连接的状态变为SYN_RCVD, 服务器把连接信息放到半连接队列里面；若已满，服务器不会将该连接的状态变为SYN_RCVD，且将该连接丢弃。
- 第二次握手：服务器返回SYN+ACK给到客户端，客户端收到SYN+ACK后，状态从SYN_SENT变为ESTABLISHED
- 第三次握手：
    - 全连接队列未满：服务器收到客户端发来的ACK, 服务端该连接的状态从SYN_RCVD变为ESTABLISHED,
      然后服务器将该连接从半连接队列里面移除，且将该连接的信息放到全连接队列里面。
    - 全连接队列已满：服务器收到客户端发来的ACK, 不会将该连接的状态从SYN_RCVD变为ESTABLISHED。

2、TCP四次挥手拆除连接

- 第一次挥手：客户端发送连接释放报文段（FIN=1，序号seq=u），并停止再发送数据，主动关闭TCP连接，进入FIN-WAIT-1状态，等待服务端的确认。
- 第二次挥手：服务端收到连接释放报文段后即发出确认报文段，（ACK=1，确认号ack=u+1，序号seq=v），服务端进入CLOSE-WAIT状态，此时的TCP处于半关闭状态，客户端到服务端的连接释放。客户端收到服务端的确认后，进入FIN-WAIT-2状态，等待服务端发出的连接释放报文段。
- 第三次挥手：服务端没有要向客户端发出的数据，服务端发出连接释放报文段（FIN=1，ACK=1，序号seq=w，确认号ack=u+1），服务端进入LAST-ACK状态，等待客户端的确认。
- 第四次挥手：客户端收到服务端的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），A进入TIME-WAIT状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间2MSL后，A才进入CLOSED状态。

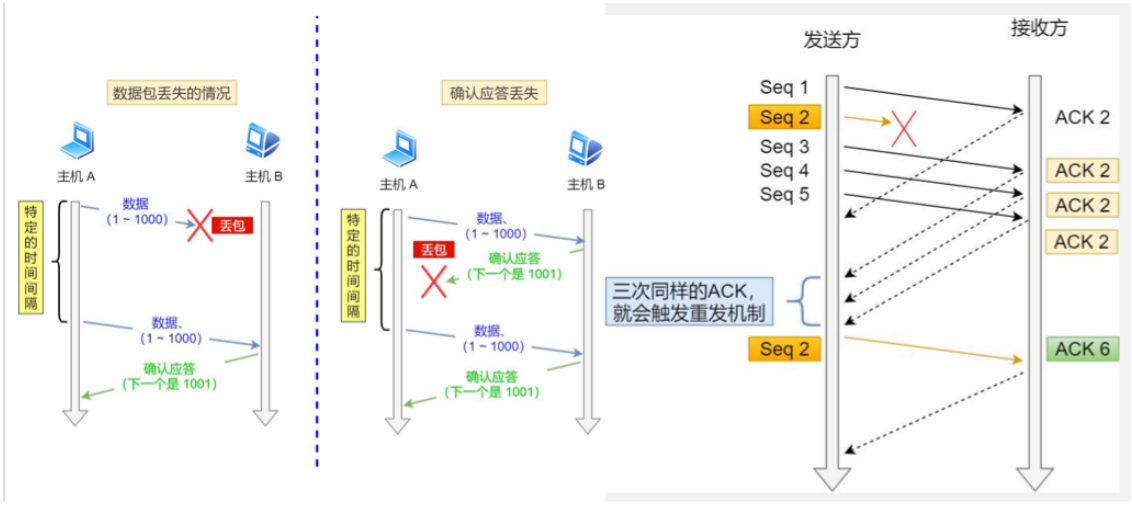


3、流量控制

- 接收方每次收到数据包，可以在发送确定报文的时候，同时告诉发送方自己的缓存区还剩余多少是空闲的，我们也把缓存区的剩余大小称之为接收窗口大小，用变量win来表示接收窗口的大小。
- 发送方收到之后，便会调整自己的发送速率，也就是调整自己发送窗口的大小，当发送方收到接收窗口的大小为0时，发送方就会停止发送数据，防止出现大量丢包情况的发生。
- 当发送方收到接受窗口 win = 0 时，这时发送方停止发送报文，并且同时开启一个定时器，每隔一段时间就发个测试报文去询问接收方，打听是否可以继续发送数据了，如果可以，接收方就告诉他此时接受窗口的大小；如果接受窗口大小还是为0，则发送方再次刷新启动定时器。

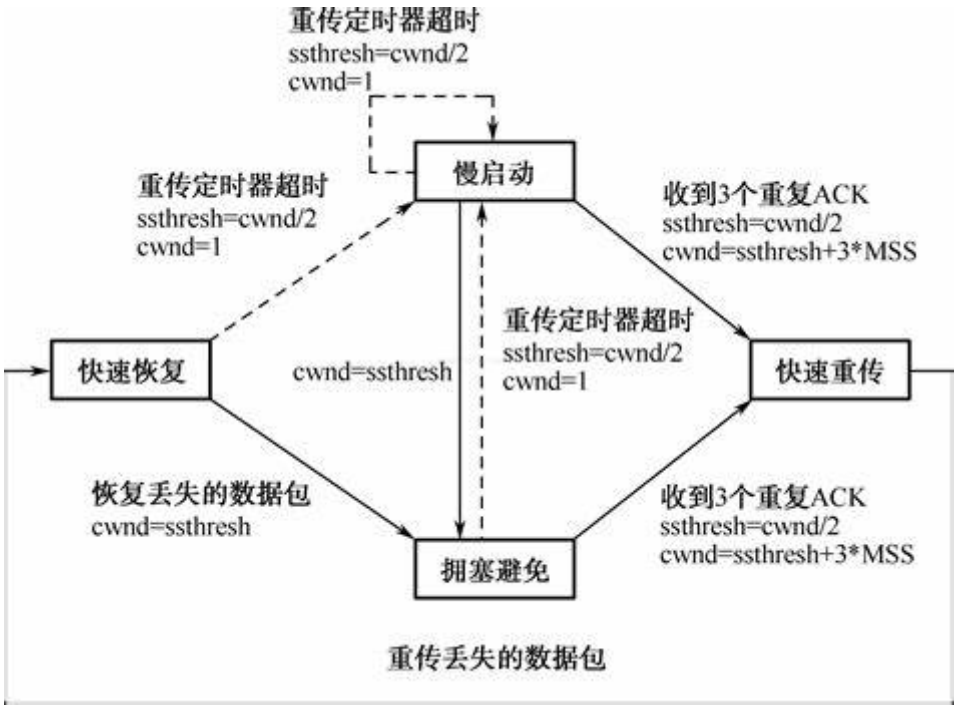
4、可靠数据传输

- 针对发送端发出的数据包的确认应答信号ACK：
  - 接收端接收到数据包后，会向发送端发送ACK号为接收到的数据包的seq号加包长的数据包；
  - 通知发送端数据包已成功接收；
  - 通知发送端发送顺序号为ACK号的数据包。
- 针对数据包到达接收端主机顺序乱掉的顺序控制：
  - 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层；
  - 接收端会丢弃重复的数据。
- 针对数据包丢失或者出现定时器超时的重发机制
  - 超时重传：在请求包发出去的时候，开启一个计时器，当计时器达到时间之后，没有收到ACK，就进行重发操作，直到达到重发上限次数或者收到ACK。
  - 快速重传：当接收方收到的数据包是不正常的序列号，那么接收方会ACK一次期望的序号，以此提醒对方重传。当发送方收到连续3条的同一个序列号的ACK（重复确认 Dup Ack），就意识到这个包丢了，从而立即重传它。



5、拥塞避免

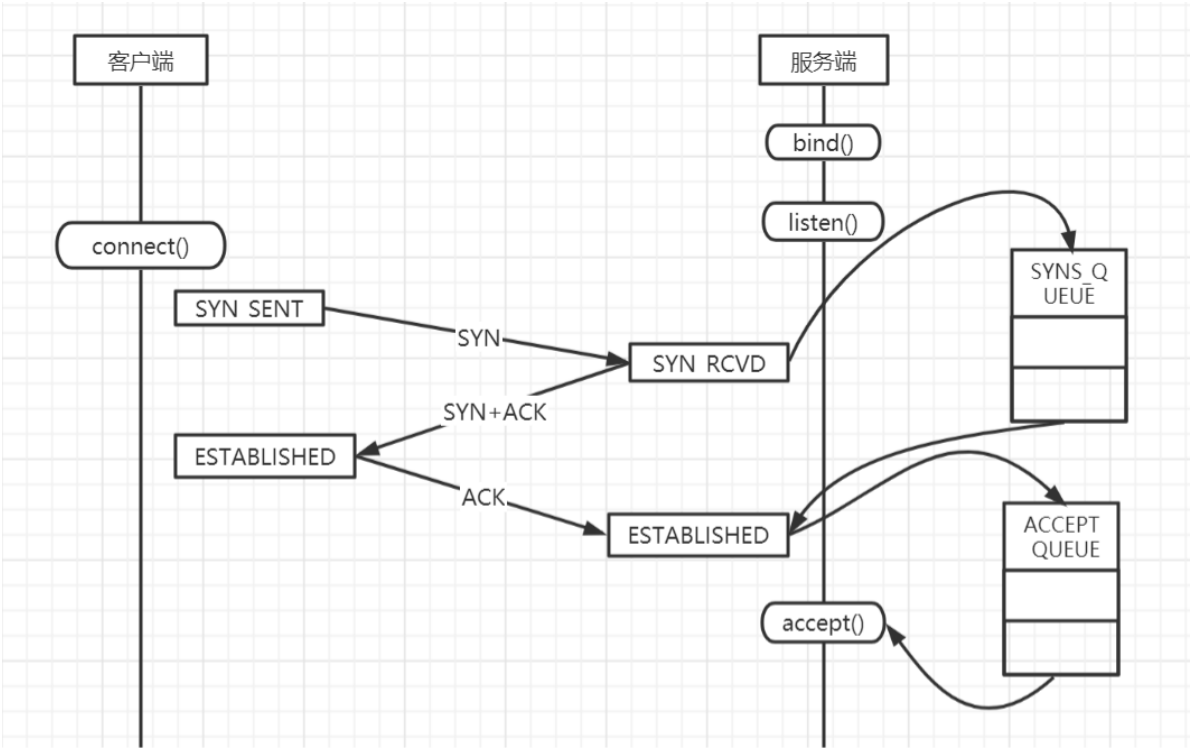
- 慢启动：最开始发送方的拥塞窗口大小为1，由小到大逐渐增大发送窗口和拥塞窗口。每经过一格传输轮次，拥塞窗口加倍。当拥塞窗口超过慢开始上限阈值，则使用拥塞避免算法，避免拥塞窗口增长过大。
- 拥塞避免：在慢开始和拥塞避免中，一旦发现网络拥堵，就把慢开始阈值设置为当前一半，并且重新设置拥塞窗口为1，重新开始增长。
- 快重传：首先要求接收方每收到一个失序的报文段后，就立即发出重复确认(为的是使发送方及早知道有报文段没有到达对方)，而不要等待自己发送数据时才进行捎带确认。
    - 把慢开始门限ssthresh 设置为 拥塞窗口cwnd 的一半；
    - 把拥塞窗口cwnd 再设置为慢开始门限ssthresh+3 的值；
    - 重新进入拥塞避免阶段。
- 快恢复：
    - 当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把慢开始门限ssthresh减半；
    - 把 cwnd 值设置为慢开始门限ssthresh 减半后的数值，然后开始执行拥塞避免算法(“ 加法增大”)，使拥塞窗口缓慢地线性增大。



四、协议实现

（一）连接管理——TCP三次握手建立连接

如图：



1、第一次握手

```
int tju_connect(){//客户端主动发送SYN报文
    //发送SYN报文,设置标志位为SYN（即0x8），随机选取seq号，ack号为0
    char* msg_syn=create_packet_buf();
    sendToLayer3(msg_syn,DEFAULT_HEADER_LEN);

    //更改socket的状态为SYN_SENT，开始第一次握手
    sock->state=SYN_SENT;

    while(sock->state!=ESTABLISHED){
    //进入establish状态前connect()函数阻塞
    }
}
```

2、第二次握手

```
int tju_handle_packet(){//在该函数中进行三次握手
    socket的状态为listen时：
    //收到客户端发送的SYN包时，被动开始三次握手，进入SYN_RCVD状态，并发送SYN+ACK包
    if(收到的包头中标志位为SYN){
        //更改socket的状态
        sock->state=SYN_RECV;

        //建立一个新的socket
        tju_tcp_t* nw_sock=tju_socket();

        nw_sock->state=SYN_RECV;//更改新创建的socket的状态
        printf("新sockt创建完成\n");
```

```
        //将新建的socket放入半连接队列，使用监听socket的hashval
        sock_queue[listen_hashval] -> syn_queue[0]=nw_sock;

        //发送SYN+ACK包，seq号随机，ack号为get_seq(pkt)+1，标志位为SYN+ACK（即0xC）
        char*msg_syn_ack=create_packet_buf();
        sendToLayer3(msg_syn_ack,DEFAULT_HEADER_LEN);
    }

}
```

3、第三次握手

```
int tju_handle_packet(){//在该函数中进行三次握手
    socket的状态为SYN_SENT时：
    //客户端期待收到来自服务端的SYN+ACK包
    if(收到的包头中标志位为SYN+ACK){
        //更改socket状态为SYN_SENT
        sock->state=ESTABLISHED;

        //发送ACK包，seq号为get_ack(pkt)，ack号为get_seq(pkt)+1，标志位为ACK（即0x4)
        char* msg_ack=create_packet_buf();
        sendToLayer3(msg_ack,DEFAULT_HEADER_LEN);
    }
}
```

4、服务端进入ESTABLISH状态

```
int tju_handle_packet(){//在该函数中进行三次握手
    socket的状态为SYN_RECV时：//期待收到来自客户端的ACK包
    if(收到的包头中标志位为ACK){

        将半连接队列中的socket取出放入全连接队列
        //更改该socket的状态为ESTABLISHED
        sock->state=ESTABLISHED;
     }
}
```

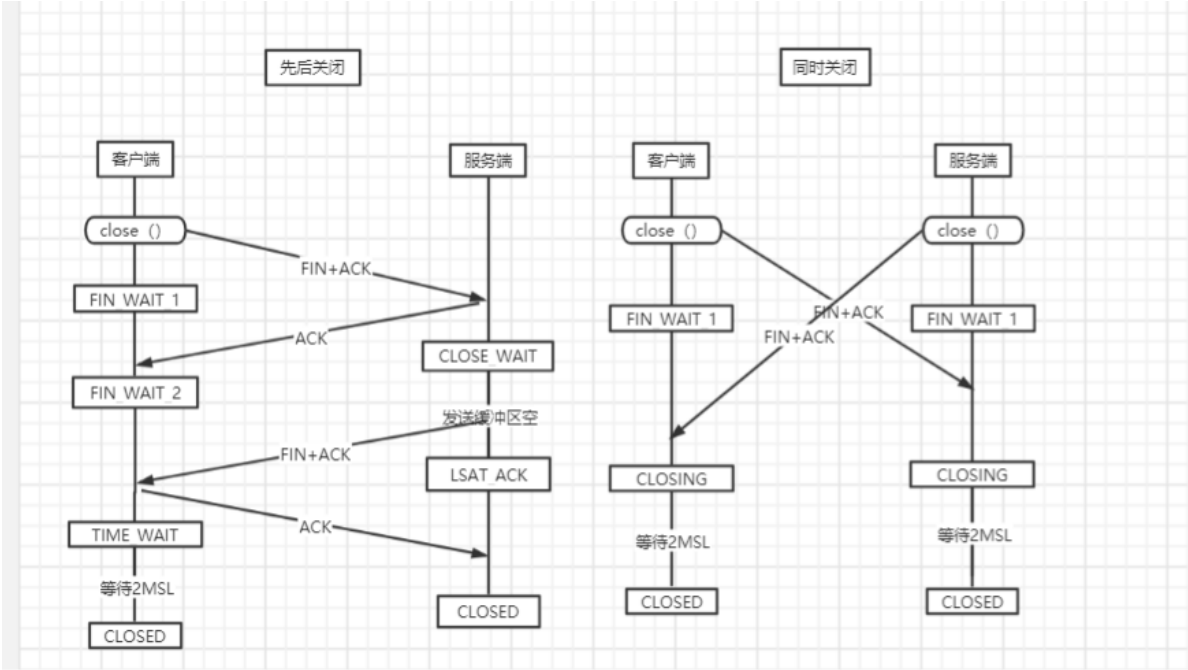
5、accept()函数

```
int accept(){
    int listen_hashval = cal_hash();

    while (1)
    {
        //从kernel维护的全连接队列中拿出已经建立号的连接，如果队列为空，则阻塞
        if(若全连接队列非空){
                拿出一个已经建立好连接的socket；
                int hashval = cal_hash();//计算该socket的hashval

                //把这个socket放入ehash中，并从全连接队列除去该连接
                established_socks[hashval] = socket;
                accept_queue[0] = NULL;

                return socket;//返回该建立好连接的socket
        }
```

```
    }
}
```

（二）连接管理——TCP四次挥手拆除连接

如图：



1、CLOSE()

```
int tju_close (){
    //进入FIN_WAIT_1状态
    sock->state = FIN_WAIT_1;

    //发送SYN+ACK包
    msg = create_packet_buf();
    sendToLayer3();
    while (1)
    {
        //等待进入CLOSED状态，关闭连接
        if(sock -> state ==CLOSED){
            return 0;
        }
    }
}
```

2、ESTABLISH状态

```
int tju_handle_packet(){
    ESTABLISH状态下：
    if(是SYN+ACK包){
        //进入CLOSE_WAIT
        sock->state = CLOSE_WAIT;
        //发送ACK包
        msg=create_packet_buf();
        sendToLayer3(msg,20);
        while(1){
```

```
            if(缓冲区为空时){
                sock->state = LAST_ACK;
                //发送ACK，进入LAST_ACK状态
                char* msg1;
                msg1 = create_packet_buf();
                sendToLayer3(msg1,20);
                break;
            }
        }
    }
}
```

3、FIN_WAIT_1状态

```
int tju_handle_packet(){
    FIN_WAIT_1状态下:
    if(收到的是ACK包){
        //进入FIN_WAIT_2状态
        sock->state = FIN_WAIT_2;
    }
    else if(收到的是FIN+ACK包){
        //客户端和服务端同时关闭，进入CLOSING状态
        sock->state = CLOSING;
        //发送ACK包,seq=get_ack(pkt)+1,ack=get_seq(pkt)+1
        msg =create_packet_buf();
        sendToLayer3(msg,20);
    }
}
```

4、FIN_WAIT_2状态

```
int tju_handle_packet(){
    FIN_WAIT_2状态下:
    if(收到FIN+ACK包){
        //进入TIME_WAIT状态
        sock->state = TIME_WAIT;
        //发送ACK包
        msg =create_packet_buf();
        sendToLayer3(msg,20);
        等待2MSL后进入CLOSED状态;
        sock->state = CLOSED;
    }
}
```

5、CLOSING状态:

```c
int tju_handle_packet(){
    CLOSING状态下:
    if(收到的是ACk包){
        //进入TIME_WAIT状态
        sock->state = TIME_WAIT;
        等待2MSL进入CLOSED状态:
        sock->state = CLOSED;
    }
}
```

6、LAST_ACK状态

```c
int tju_handle_packet(){
    LAST_ACK状态下:
    if(收到的是ACK包){
        进入CLOSED状态:
        sock->state = CLOSED;
    }
}
```

（三）发送数据

1、将数据放入发送缓冲区

```c
int tju_send(){
    //将待发送数据拆分放入发送缓冲区

    if(/*socket状态不是ESTABLISH或者CLOSE_WAIT*/){//其它状态不会在发送信息
        return -1;
    }

    int i=0;
    while(i<len){//信息没有完全发送

        char* send_data=data+i;
        int send_len=len-i > MAX_DLEN ? MAX_DLEN : len-i;//每个数据包携带的长度不超过MAX_DLEN
        i+=send_len;
        if(/*缓冲区未满*/){
            //放入发送缓冲
            add_to_send_buf();
        }
        else{
            printf("\n发送缓冲区满\n");
        }
    }
    return 0;
}
```

```c
void add_to_send_buf(){

    pthread_mutex_lock(&sock->send_lock);
```

```
    初始化缓冲区节点;

    if(缓冲区为空){
        //将该节点加入发送缓冲区队列
        sock->sending_buf->head=nw_node;
        sock->sending_buf->tail=nw_node;
        //该节点加入待发送
        sock->send_head=sock->sending_buf->head;
        //更新缓冲区长度
        sock->sending_len+=send_len;
    }
    else{
        //将该节点加入缓冲区队尾，并将队尾设置为该节点
        sock->sending_buf->tail->next=nw_node;
        nw_node->prev=sock->sending_buf->tail;
        sock->sending_buf->tail=nw_node;

        if(待发送节点为空){
            sock->send_head=nw_node;//设置为该节点
        }
        sock->sending_len+=send_len;
    }
    pthread_mutex_unlock(&sock->send_lock);

    return;
}
```

2、数据发送线程

```
void * send_thread(){
    //在connect()和accept()函数建立连接后调用该线程，循环发送缓冲区中的内容
    while(1)
    {
        if(待发送节点sock->send_head不为空){
            while(pthread_mutex_lock(&(nw_sock->send_lock))!=0);
            if(发送窗口未满){
                //获取发送时间
                gettimeofday();
                //打包发送
                char* msg=create_packet_buf();
                sendToLayer3();
                更新socket中的待发送节点，发送窗口的nextseq和wait_for_ack;
                //创建计时器
                nw_head->skb_timer=creat_timer();
            }
            else{//发送缓冲区已满
                while(发送缓冲区满){}//阻塞
            }
            pthread_mutex_unlock(&(nw_sock -> send_lock));
        }
    }
}
```

（四）接收数据

1、将接收到的数据包放入接收窗口

```
void add_to_wnd_recv(){

    if(接收窗口为空){
        放入接收窗口;
        sock->wnd_recv->recv_len+=nw_node->len;

    }
    else{//接收窗口不空
        recv_skb_node* tmp=recv_wnd->head;
        if(收到的seq号比接收窗口seq号小){
            放入接收窗口头部;
        }
        else{
            while(收到的seq号比接收窗口seq号小){
                tmp=tmp->next;
            }
            //按顺序找到位置
            if(nw_node->seq==tmp->seq){
                printf("\n重复数据包，丢弃\n");
            }
            else{
                //插入接收队列
                nw_node->prev=tmp->prev;
                nw_node->next=tmp;
                tmp->prev=nw_node;
            }
        }
    }
}
```

2、将接受窗口数据包放入接收缓冲区

```
void wnd_to_buf(){
    while (pthread_mutex_lock(&(sock->recv_lock)) != 0); // 加锁

    if(接收缓冲区为空){
        放入缓冲区队列头部;
    }
    else{
        放入缓冲区队列尾部;
    }
    pthread_mutex_unlock(&(sock->recv_lock));

}
```

3、从缓冲区读数据

```
int tju_recv(){

    while(缓冲区无数据){}//阻塞

    while(pthread_mutex_lock(&(sock->recv_lock)) != 0); // 加锁
```

```
    recv_skb_node* nw_node=sock->received_buf->head;
    if(当前节点长度小于读取长度){
        memcpy();//读取节点数据
        now_len+=nw_node->len;
        //缓冲区推进
        head=head->next;
        received_len-=nw_node->len;
    }
    else{//不能完全读取节点数据
        int read_len=now_len+nw_node->len-len;//读取长度
        memcpy();
        //将该节点剩余数据放入新节点
        char * nw_buffer=malloc(nw_node->len-read_len);
        memcpy();
        //将新节点放入缓冲区头
        nw_node->next=head->next;
        head=nw_node;
        //更新缓冲区长度
        sock->received_len-=read_len;
        now_len+=read_len;
    }
    free();//释放资源
    pthread_mutex_unlock(&(sock->recv_lock)); // 解锁
    return 0;
}
```

（五）处理数据包

1、处理收到的数据包

```
int tju_handle_packet(){
    ESTABLISH状态下收到的是数据包：

    初始化接收缓冲区节点；

    if(如果接收窗口或接收缓冲区已满){
        //通知发送方接收窗口为0，advertised_window=0
        char* nw_pkt = create_packet_buf();
        sendToLayer3();
    }
    else{
        if(收到的数据包的seq是接收窗口期望的seq)
        {
            //放入接收窗口
            add_to_wnd_recv(sock,nw_node);
            //将接收窗口中按序排好的数据包放入接收缓冲区
            while(数据包seq按序){
                //放入接收缓冲区
                wnd_to_buf();
                //从接收窗口中拿出
                sock->wnd_recv->recv_len-=nw_node->len;
                sock->wnd_recv->window->head=sock->wnd_recv->window->head->next;
                //更新接收窗口期望seq,实现累计确认
                sock->wnd_recv->expect_seq+=nw_node->len+DEFAULT_HEADER_LEN;
            }
        }
```

```
        else{
            //乱序数据包，只放入接收窗口，而不更新expect_seq
            add_to_wnd_recv();
        }
        //发送ACK通知发送方期望的seq
        char* nw_pkt = create_packet_buf();
        sendToLayer3(nw_pkt, DEFAULT_HEADER_LEN);
    }
}
```

2、处理收到的ACK

```
int tju_handle_packet(){
    ESTABLISH状态下收到的是ACK包：

    --流量控制：
    get_advertised_window(pkt);//获取ACK包中的advertised_window
    更新发送方窗口大小；
    if(窗口大小为0){
        creat_timer(10000,persist);//创建坚持计时器，发送TEST测试包
    }
    else{//窗口不为0
        if(如果坚持计时器在运行){
            释放资源；
        }
    }
    --重复ACK：
    if(收到的是重复ACK){
        sock->wnd_send->ack_cnt++;
        if(收到三次重复ACK){
            retransmit(sock);//快速重传
        }
    }
    else{
        --计算rto：
        gettimeofday();//获取接收时间
        cal_rto();//计算rto

        --窗口推进：
        int add_len = ack - sock->wnd_send->base;
        while(add_len > 0)
        {
            从发送缓冲区头开始，寻找已经被确认的数据包：
            sock->sending_len -= send_node->len;//更新缓冲区长度
            sock->wnd_send->base += send_node->len+DEFAULT_HEADER_LEN;//更新已确认
数据包号
            sock->sending_buf->head = send_node->next;//窗口推进
            free();//释放资源
        }
    }
}
```

3、处理收到的窗口探测包

```
int tju_handle_packet(){
    ESTABLISH状态下收到的是数据包：
    //设置advertised_window为TCP_RECVWN_SIZE-sock->wnd_recv->recv_len，通知发送方窗口大小
    char* pkt = create_packet_buf();
    sendToLayer3();
}
```

（六）计时器实现

1、初始化setitimer函数

```
//设置每过INITIAL_INTERVAL时长向handle_signal函数发送SIGALRM信号
struct itimerval timeval;
timeval.it_value.tv_sec = 0;
timeval.it_value.tv_usec = INITIAL_INTERVAL;
timeval.it_interval.tv_sec = 0;
timeval.it_interval.tv_usec = INITIAL_INTERVAL;
setitimer(ITIMER_REAL , &timeval , NULL );
signal(SIGALRM , signalhandler );
```

2、创建计时器

```
timer *creat_timer(){
    timer* nw_timer=(timer*)malloc(sizeof(timer));
    设置超时时长为计算的RTO时长;
    nw_timer->func=func;//设置超时执行的功能
    return nw_timer;
}
```

3、处理SIGALRLM信号

```
void handle_signal(){
    遍历所有socket，调用time_clock，对socket的每个发送窗口等待确认的数据包的计时器做出更改;
    return ;
}
void time_clock(){
    if(如果persist_timer不空){
        now_time-=INITIAL_INTERVAL;//更新计时器时间
        if(now_time到0){
            调用persist()函数
            now_time=set_time;//重置计时器，重新计时
        }
    }
    if(发送窗口不空){
        while(节点不空，且每到待发送节点(即已发送等待确认节点)){
            if(计时器非空){
                now_time-=INITIAL_INTERVAL;//更新计时器时间
                if(超时){
                    调用retransmit()函数重传;
                    now_time=set_time;//重置计时器，重新计时
                }
                if(nw_head->next!=NULL){
                    nw_head=nw_head->next;//遍历已发送等待确认节点
```

```
                }
                else{
                    break;
                }
            }

        }
    }
}
```

（七）retransmit和persist功能实现

1、重传函数

```
void retransmit(){
    //重传发送缓冲区头的数据包
    skb_node* nw_node=sock->sending_buf->head;
    char* pkt=create_packet_buf();
    sendToLayer3();
}
```

2、persist函数

```
void persist(){
    char* pkt=create_packet_buf();//设置标志位为TEST
    sendToLayer3();
}
```

（八）计算RTO

```
void cal_rto(){
    skb_node* nw_node=sock->sending_buf->head;
    while(nw_node!=NULL){
        从发送缓冲区头开始遍历节点找到该ACK确认的数据包;
    }
    int rtt_time=(recv_time.tv_sec-nw_node->send_time.tv_sec)*1000000+
(recv_time.tv_usec-nw_node->send_time.tv_usec);//计算RTT市场

    if(estmated_rtt为初始化rtt){
        estmated_rtt=rtt_time;
        rtt_var=rtt_time/2;
    }
    else{
        estmated_rtt= 0.875 * sock->wnd_send->estmated_rtt + 0.125 * rtt_time;
        rtt_var = 0.75 * sock->wnd_send->rtt_var + 0.25 * abs(sock->wnd_send-
>estmated_rtt - rtt_time);

    }
    //设置rto
    int rto = sock->wnd_send->estmated_rtt + 4 * sock->wnd_send->rtt_var;
}
```

（九）拥塞控制

```
收到的不是重复的ACK：
switch (sock->wnd_send->congestion_status)
{
    case 慢启动:
        cwnd=2 * cwnd;
        if(到达预设的ssthresh){
            congestion_status=CONGESTION_AVOIDANCE;//进入拥塞避免
        }
        break;
    case 拥塞避免:
        cwnd += MAX_DLEN;//每过一个rtt,增加一个MAX_DLEN
        break;
    case 慢恢复:
        ssthresh =cwnd / 2;
        cwnd = ssthresh;
        congestion_status = CONGESTION_AVOIDANCE;//进入拥塞避免
        break;

    default:
        break;
}
收到三个重复ACK：
congestion_status = FAST_RECOVERY;//进入快恢复
超时：
if(不是快恢复状态){
    ssthresh = cwnd / 2 ;//重设ssthresh
    cwnd = MAX_DLEN;
    congestion_status = SLOW_START;//进入慢启动
}
```

五、实验结果及分析

（一）连接管理

- 成功三次握手建立连接；
- 成功四次挥手拆除连接，并能成功解决客户端和服务端同时关闭的情况。

（二）可靠传输

- 成功实现发送接收缓冲区，成功传递信息；
- 成功实现滑动窗口机制，接收到ACK后推荐发送窗口：

```
RTT为2634098

RTO估计值为4299167
```

- 成功实现超时重传和快速重传机制：

```
超时

重传一个数据包test message37
```

```
重复ACK
收到一个ACK包720
重复ACK
收到一个ACK包720
重复ACK
收到一个ACK包720
快速重传
重传一个数据包test message20
```

- 成功实现累计确认机制（乱序的包到达后，将后面提前到达排好序的包一同确认）：

```
放入接收窗口完成test message20
,720
nw_head=720,nw_head->len=16,nw_head->next->seq=756
将节点放入缓冲区
接收缓冲区不空
放入接收缓冲区完成test message20
,720,缓冲区长16
nw_head=756,nw_head->len=16,nw_head->next->seq=792
将节点放入缓冲区
接收缓冲区不空
放入接收缓冲区完成test message21
,756,缓冲区长32
nw_head=792,nw_head->len=16,nw_head->next->seq=828
将节点放入缓冲区
接收缓冲区不空
放入接收缓冲区完成test message22
,792,缓冲区长48
nw_head=828,nw_head->len=16,nw_head->next->seq=864
```

- 成功实现计算RTO，并据此设置超时时长：

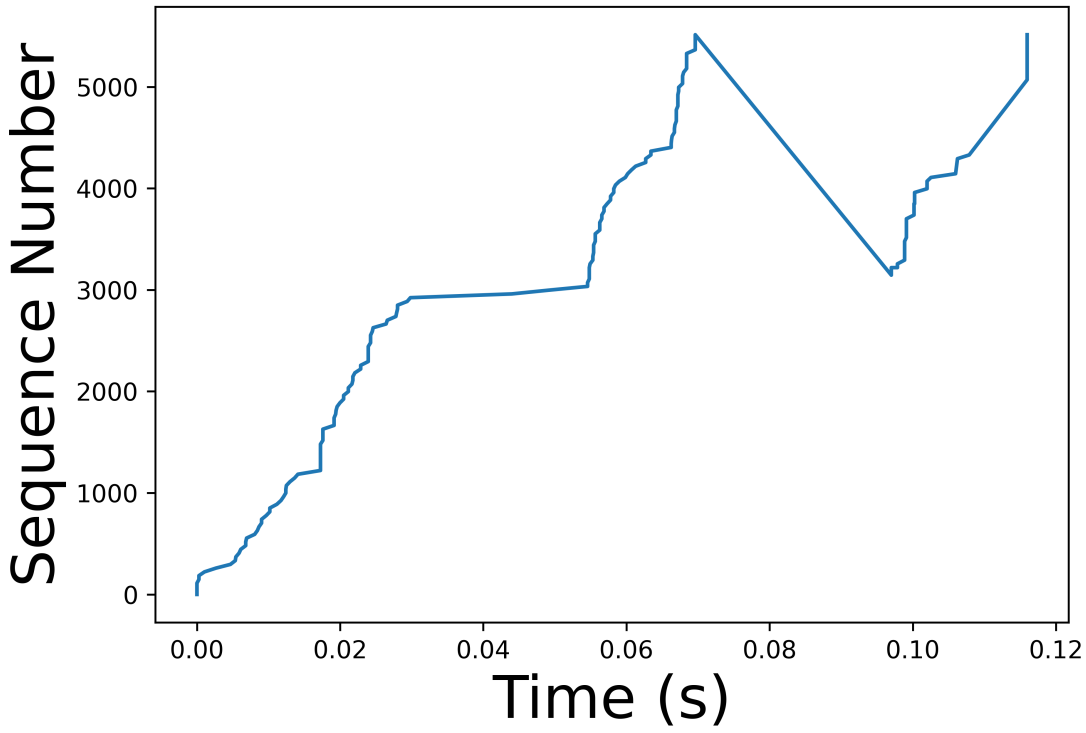
```
RTT为2634098

RTO估计值为4299167
```

（三）流量控制

- 成功在ACK包中发送接收窗口大小信息，并能在窗口为0时发送TEST检测包，并创建坚持计时器。

（四）拥塞控制

- 成功实现慢启动、拥塞避免、快恢复算法：



- 0.00~0.02s在慢启动状态，此后进入拥塞避免状态，0.07s收到重复ACK，进入快恢复，cwnd减半后进入再次进入拥塞避免状态；
- 多次抓包后结果仍旧不太理想，推测可能是初始超时时长设置不合理，导致过多包重传。

六、个人总结

- 在协议设计过程中，由于没有很好的理解TCP的整体工作流程，没有整体的认知，导致起初设计时各部分功能不能很好的实现；
- 实现计时器的过程中遇到问题，最终使用`setitimer()`函数解决；
- 结构体资源释放问题：缓冲区节点内存释放时，不能只调用`free()`函数，还需要将结构体内的指针free掉，即结构体资源释放的时候，从里向外，先释放成员指针，再释放结构体指针；
- 总体来说，TCP的各部分功能实现的较为成功，在实践中对TCP协议的协议规则有了更深的理解，并且对协议设计及实现的流程有了更好的认识。