TCP网络传输实验

一、实验内容

TCP网络传输实验，要求在已有代码基础和给定网络拓扑和节点配置上，实现TCP连接管理、数据传输、丢包恢复、拥塞控制等功能，使得两个协议栈之间能够进行可靠数据传输，并根据丢包情况按照TCP newReno规则调整自己的拥塞窗口大小。

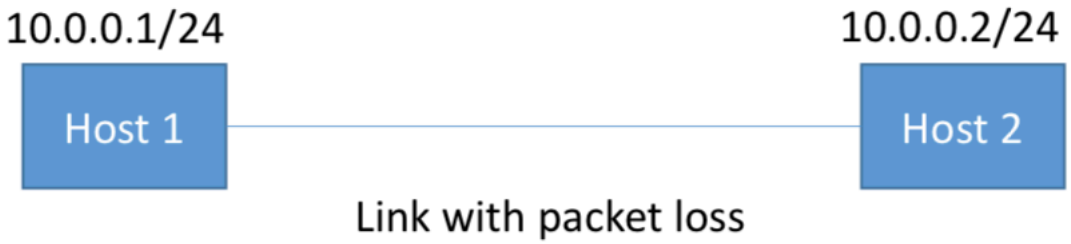

图1-1 实验网络拓扑结构

实验主要内容：

- TCP协议数据结构实现
- 无丢包环境下
  - TCP连接管理
  - 数据传输
- 在有丢包环境下
  - TCP连接管理和数据传输
  - TCP拥塞控制

二、实验环境

- 操作系统：Ubuntu 16.04 64位
- 实验环境：mininet
- 编程语言：c，python2

三、实验过程

1、实验代码结构说明

实验代码目录结构如下图所示，其中，字体标蓝的函数表示需要补充的函数。

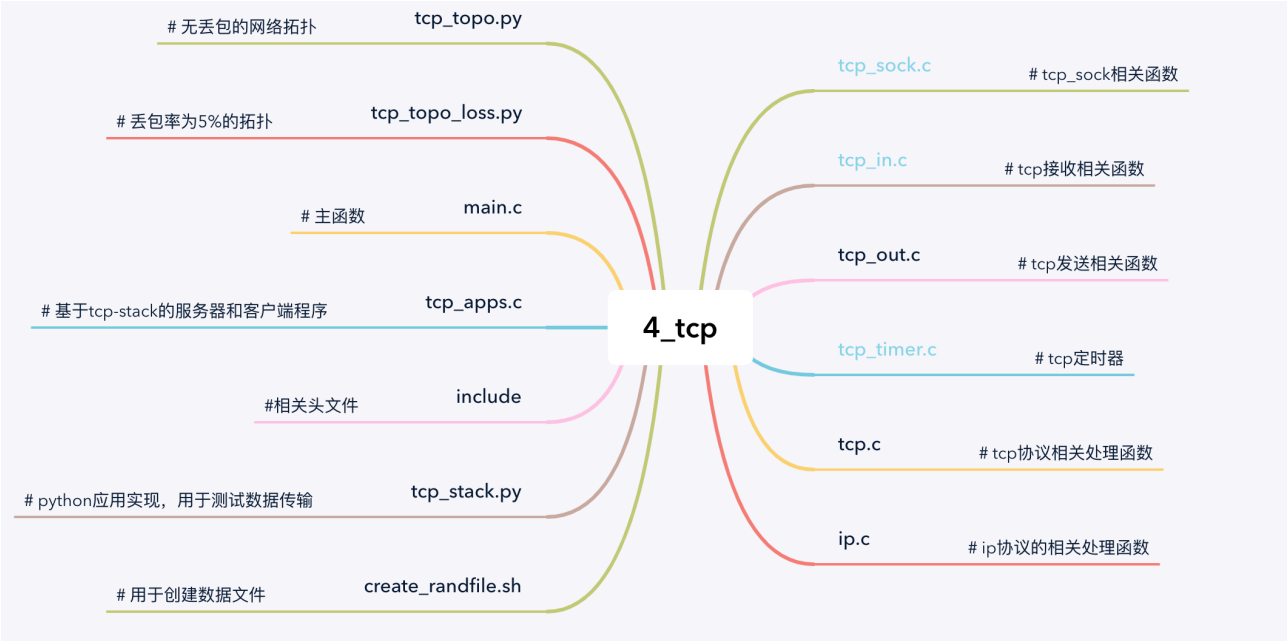

图3-1 实验代码目录结构

2、TCP协议数据结构实现

struct tcp_sock 是Socket维护TCP连接信息和数据传输控制的核心数据结构。struct tcp_sock 定义在 include/tcp_sock.h 下。它主要包括四个部分：

- 连接双方的IP地址和端口信息
- 当前的状态（TCP状态）
- 收发数据序列号
- TCP拥塞控制参数

（1）通过数据包信息查找对应的Socket

根据连接所在的不同阶段，Socket绑定不同的元组信息。协议栈维护listen_table和established_table 两个hash表，来分别组织只绑定源地址、端口的socket和绑定四元组的socket。

```
struct tcp_hash_table {
      struct list_head established_table[TCP_HASH_SIZE];
      struct list_head listen_table[TCP_HASH_SIZE];
      struct list_head bind_table[TCP_HASH_SIZE];
};[1]
```

tcp_sock.c 中 struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport) 函数，用以查找established_table中匹配四元组的socket。具体代码思路如下：

```
struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16
sport, u16 dport)
{
      int hash;
      struct list_head *list;
      hash = tcp_hash_function(saddr, daddr, sport, dport);
      list = &tcp_established_sock_table[hash];

      struct tcp_sock *tsk;
      list_for_each_entry(tsk, list, hash_list) {
            if (tsk->sk_sip == saddr &&
                  tsk->sk_dip == daddr &&
                  tsk->sk_sport == sport &&
                  tsk->sk_dport == dport)
                  return tsk;
      }

      return NULL;
}
```

[1] 注：文档中，绿色标注为补充代码部分；蓝色为需要补充的函数名称；黄色为补充函数的关联文件/函数；紫色为特殊数据结构。

首先利用tcp_hash.h 中 tcp_hash_function(u32 saddr, u32 daddr, u16 sport, u16 dport)函数将已知四元组hash，找到established_table中该hash值的list，利用list.h 中 list_for_each_entry(pos, head, member) 找到源、目的ip、端口向匹配的tcp_sock *tmp 返回；若没有找到，返回NULL。

struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport) 在 tcp_listen_sock_table中，只需将源端口带入，源ip、目的ip和端口均为0。其他部分同上。

```
struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
{
        int hash;
        struct list_head *list;
        hash = tcp_hash_function(0, 0, sport, 0);
        list = &tcp_listen_sock_table[hash];
        struct tcp_sock *tsk;
        list_for_each_entry(tsk, list, hash_list) {
                if (tsk->sk_sport == sport)
                        return tsk;
        }

        return NULL;
}
```

3、TCP建立连接

对于被动建立连接一方，建立连接过程如下：

- 申请占用一个端口号（bind操作）
- 监听该端口（listen操作）
- 收到SYN数据包，状态转移为TCP_SYN_RECV（accept操作）

- 回复ACK并发送SYN数据包
- 收到ACK数据包，状态转移为TCP_ESTABLISHED

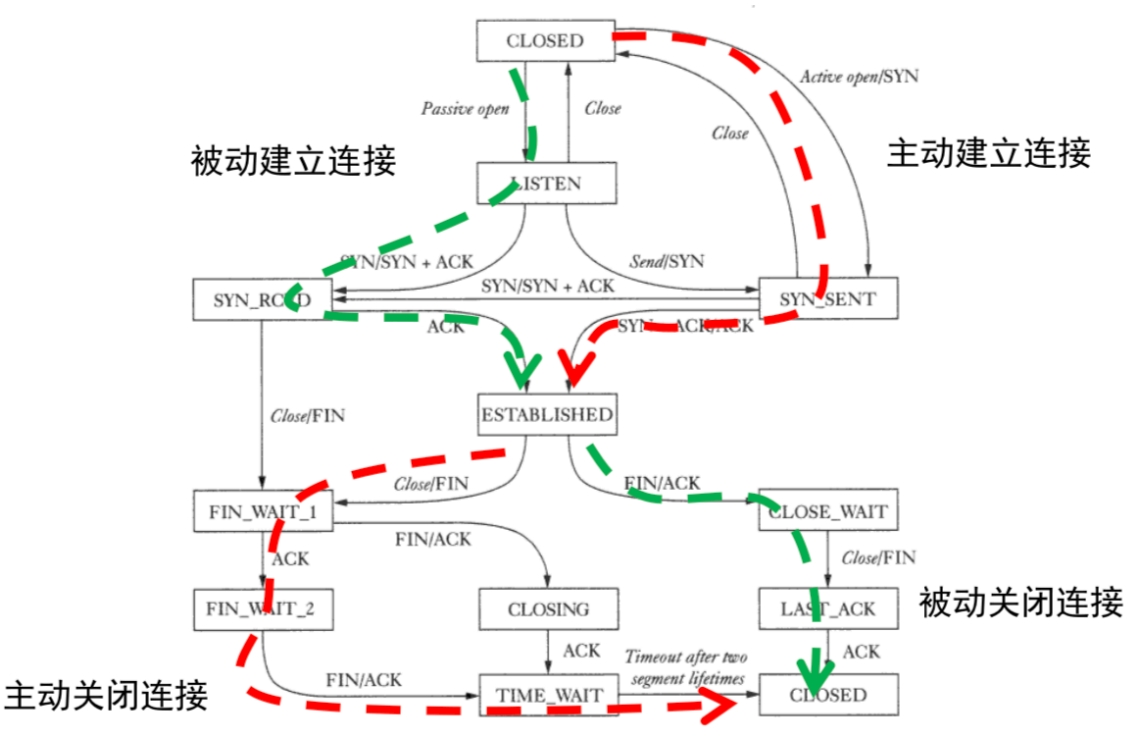

图3-2 TCP状态迁移图

对于主动建立连接一方，建立连接过程如下：

- 发送目的端口为SYN数据包，状态转为TCP_SYN_SENT状态
- 收到SYN数据包（设置TCP_ACK标志位）
- 回复ACK数据包，状态转移为TCP_ESTABLISHED

(1) 被动方监听端口

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{

    int err = 0;
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
```

```
        err = tcp_hash(tsk);
        return err;
}
```

设置tcp_sock中 backlog（挂起连接的最大数量请求），并且切换TCP_STATE，并将tcp_sock hash 到listen_table上。

(2) 被动方accept操作

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
        struct tcp_sock* csk = NULL;
        if (!list_empty(&tsk->accept_queue))
        {
                csk = tcp_sock_accept_dequeue(tsk);
        }
        else
        {
                log(DEBUG, "waiting for incoming connection request!");
                sleep_on(tsk->wait_accept);
                csk = tcp_sock_accept_dequeue(tsk);
        }

        return csk;
}
```

当被动连接建立的parent socket收到SYN数据包后，会产生一个child socket，放到parent socket 的listen_queue队列中。当接收到三次握手中的最后一个包时，在listen_queue中的child socket会放到accept_queue中，等待应用程序读取。如果accept_queue不是empty，则弹出第一个tcp_sock并接受它，否则，在wait_accept上休眠以获取传入的连接请求。

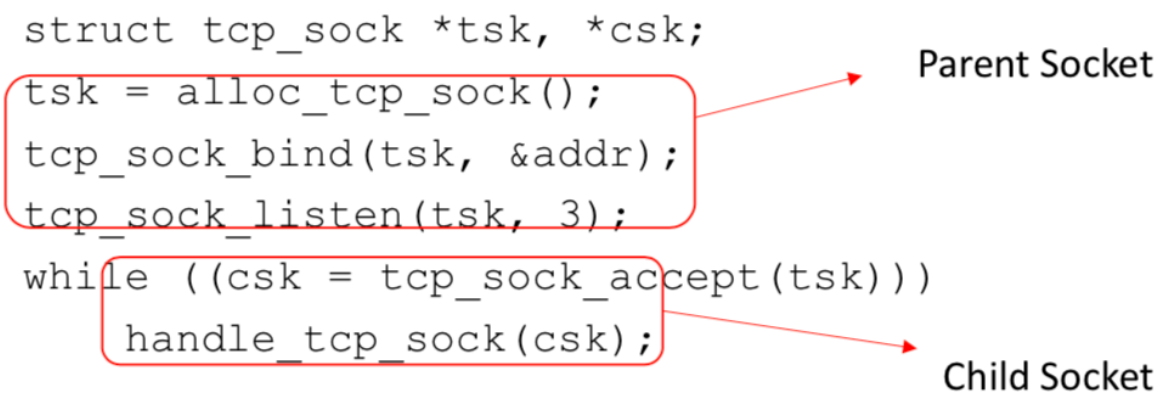

图3-3 被动建立连接一方的处理流程

在此需要说明，child socket在被tcp_socket_accept返回之前，需要保存在parent socket的队列中：

- listen_queue：未完成三次握手的child socket
- accept_queue：已完成三次握手的child socket

（3）主动方建立连接过程

```
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
        fiface_info_t* iface;
        iface = list_entry(instance->iface_list.next, typeof(*iface), list);
        tsk->sk_dip = htonl(skaddr->ip);
        tsk->sk_dport = htons(skaddr->port);
        tsk->sk_sip = iface->ip;
        tsk->sk_sport = htons(tcp_get_port());
        tcp_bind_hash(tsk);
        tcp_set_state(tsk, TCP_SYN_SENT);
        tcp_hash(tsk);
        tcp_send_control_packet(tsk, TCP_SYN);

        sleep_on(tsk->wait_connect);
        return 0;
}
```

int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)

实现连接到skaddr指定的远程tcp sock。首先初始化四个关键元组（sip，sport，dip，dport），在instance->iface_list找到自己的ip，由tcp_get_port()函数获得一个空闲端口。将tcp sock散列为bind_table后，发送SYN数据包，切换到TCP_SYN_SENT状态，等待传入；并在wait_connect上休眠的SYN数据包。如果对等体的SYN数据包到达，则通知此功能，表示建立连接。

4、断开连接

对于主动关闭连接的一方：

- 发送FIN包，进入TCP_FIN_WAIT_1状态
- 收到FIN对应的ACK包，进入TCP_FIN_WAIT_2状态
- 收到对方发送的FIN包，回复ACK，进入TCP_TIME_WAIT状态
- 等待2*MSL时间，进入TCP_CLOSED状态，连接结束
- 需要实现定时器线程，定期扫描，适时结束TCP_TIME_WAIT状态的流

对于被动关闭的一方：

- 收到FIN包，回复相应ACK，进入TCP_CLOSE_WAIT状态
- 当自己没有待发送数据时，发送FIN包，进入TCP_LAST_ACK状态
- 收到FIN包对应的ACK，进入TCP_CLOSED状态，连接结束

```
void tcp_sock_close(struct tcp_sock *tsk)
{
	switch (tsk->state)
	{
	case TCP_CLOSED:
		break;
	case TCP_LISTEN:
		tcp_sock_accept_enqueue(tsk);
		tcp_unhash(tsk);
		tcp_set_state(tsk, TCP_CLOSED);
```

```
            break;
        case TCP_SYN_RECV:
            break;
        case TCP_SYN_SENT:
            break;
        case TCP_ESTABLISHED:
            tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            break;
        case TCP_CLOSE_WAIT:
            tcp_send_control_packet(tsk, TCP_FIN);
            tcp_set_state(tsk, TCP_LAST_ACK);
            break;
        }
}
```

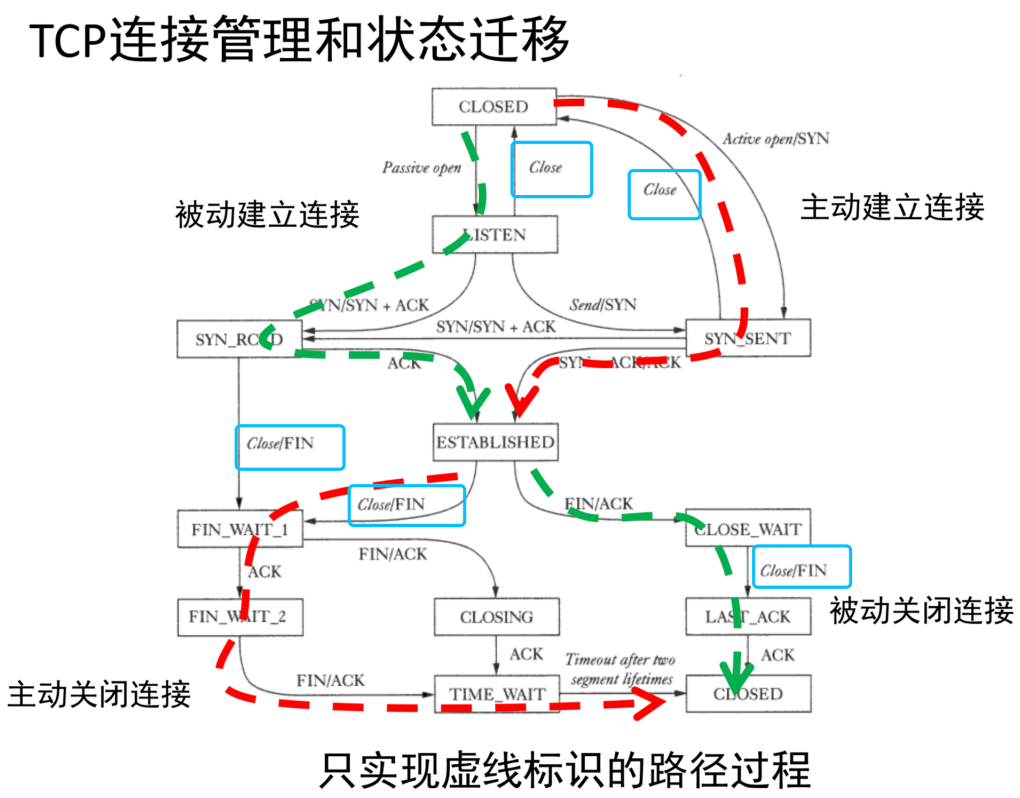

图3-4 蓝框为tcp_sock_closed 函数实现思路

5、TCP接收数据包后的处理流程

（1）数据读写部分

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int size)
{
	int read_size = 0;
	if (ring_buffer_empty(tsk->rcv_buf))
	{
		sleep_on(tsk->wait_recv);
	}
	read_size = read_ring_buffer(tsk->rcv_buf, buf, size);
	tsk->rcv_wnd += read_size;
	log(DEBUG, "tcp_sock read %d bytes of data.", read_size);

	return read_size;

	return 0;
}
```

使用该函数接收socket发送过来的数据。若环形缓冲区为空，则等待wait_recv，唤醒后读取数据、调整窗口并返回。若环形缓冲区不为空，则直接调整窗口并返回数据。

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int size)
{
	tcp_send_data(tsk, buf, len);
	return 0;
}
```

使用该函数发送数据。直接调用tcp_send_data函数发送数据。在tcp_out.c中补充函数tcp_send_data，改函数实现将buf内的值写入packet，然后利用tcp_send_packet函数实现packet的发送。

```
int tcp_send_data(struct tcp_sock *tsk, char *buf, int len)
{
	tsk->snd_wnd = len;
```

```
    int pkt_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE +
tsk->snd_wnd;
    char *packet = (char*)malloc(pkt_len);
    if (packet == NULL)
        return -1;
    char* tcp_data = packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
TCP_BASE_HDR_SIZE;
    memcpy(tcp_data, buf, tsk->snd_wnd);
    tcp_send_packet(tsk, packet, pkt_len);
    return 0;
}
```

（2）处理接收到的数据包

对接收到的数据包，需要注意：

- 检查TCP校验和是否正确
- 检查是否为RST包，如果是，直接结束连接
- 检查是否为SYN包，如果是，进行建立连接管理
- 检查ack字段，对方是否确认了新的数据
- 本次实验中只有SYN和FIN包会确认新数据
- 检查是否为FIN包，如果是，进行断开连接管理

```
void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
    char cb_flags[32];
    tcp_copy_flags_to_str(cb->flags, cb_flags);
    log(DEBUG, "recived tcp packet %s", cb_flags);
    switch (tsk->state)
    {
    case TCP_CLOSED:
        tcp_state_closed(tsk, cb, packet);
        return;
        break;
```

```
case TCP_LISTEN:
        tcp_state_listen(tsk, cb, packet);
        return;
        break;
case TCP_SYN_SENT:
        tcp_state_syn_sent(tsk, cb, packet);
        return;
        break;
default:
        break;
}

if (!is_tcp_seq_valid(tsk, cb))
{
        // drop
        log(ERROR, "received tcp packet with invalid seq, drop it.");
        return;
}

if (cb->flags & TCP_RST)
{
        //close this connection, and release the resources of this tcp sock
        tcp_set_state(tsk, TCP_CLOSED);
        tcp_unhash(tsk);
        return;
}

if (cb->flags & TCP_SYN)
{
        //reply with TCP_RST and close this connection
        tcp_send_reset(cb);
        tcp_set_state(tsk, TCP_CLOSED);
```

```
            tcp_unhash(tsk);
            return;
}

if (!(cb->flags & TCP_ACK))
{
            //drop
            log(ERROR, "received tcp packet without ack, drop it.");
            return;
}
//process the ack of the packet
if (tsk->state == TCP_SYN_RECV)
{
            tcp_state_syn_recv(tsk, cb, packet);
            return;
}
if (tsk->state == TCP_FIN_WAIT_1)
{
            tcp_set_state(tsk, TCP_FIN_WAIT_2);
            return;
}
if (tsk->state == TCP_LAST_ACK)
{
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
            return;
}
if (tsk->state == TCP_FIN_WAIT_2)
{
            if (cb->flags != (TCP_FIN | TCP_ACK))
            {
                        //drop
```

```
                log(ERROR, "received tcp packet without FIN|ACK, drop it.");
                return;
        }
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);
        // start a timer
        tcp_set_timewait_timer(tsk);
        tcp_set_state(tsk, TCP_TIME_WAIT);
        return;
}
//update rcv_wnd
tsk->rcv_wnd -= cb->pl_len;
//update snd_wnd
tcp_update_window_safe(tsk, cb);
//recive data
if (cb->pl_len > 0)
        tcp_recv_data(tsk, cb, packet);

if (cb->flags & TCP_FIN)
{
        //update the TCP_STATE accordingly
        tcp_set_state(tsk, TCP_CLOSE_WAIT);
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_send_control_packet(tsk, TCP_FIN | TCP_ACK);
        tcp_set_state(tsk, TCP_LAST_ACK);
        return;
}

//reply with TCP_ACK if the connection is alive
if (cb->flags != TCP_ACK)
{
```

```
                tsk->rcv_nxt = cb->seq_end;
                tcp_send_control_packet(tsk, TCP_ACK);
        }
}
```

首先，判断socket处于的状态，如果socket处于TCP_CLOSED状态，则交给tcp_state_closed函数处理，发送RST数据包；当socket处于TCP_LISTEN状态，则交给tcp_state_listen函数处理。tcp_state_listen函数首先创建一个child socket，接着对client发送SYN/ACK，并且将child socket hash 到established_table。详细过程见tcp_state_listen。当socket处于TCP_SYN_SENT状态，tcp_state_syn_sent函数进行处理处理。tcp_state_syn_sent函数原理：首先检查收到的数据包，如果非SYN/ACK，则回复RST；否则，回复ACK完成三次握手，转为TCP_ESTABLISHED状态。

接着利用is_tcp_seq_valid函数验证报文的序列号是否正确。若不正确，直接丢弃这个包。检测数据包的标志位，如果收到RST数据包，不必发送ACK包来确认，直接进入closed状态。若收到SYN数据包，因为前面函数进行判断，所以这个时候收到的SYN数据包为异常包，回复RST关闭连接。若收到ACK不置1的数据包，直接丢弃。

如果socket处于TCP_SYN_RECV状态，调用tcp_state_syn_recv函数处理：从parent的监听队列中删除自己，添加到parent的接受队列，由于此时自己已经建立连接，wait_accept parent。当socket处在TCP_FIN_WAIT_1状态，将socket状态转换到TCP_FIN_WAIT_2状态；当socket处于TCP_LAST_ACK，则将socket状态转换到TCP_CLOSED状态，这个时候需要将它从established_table中删除。

当socket处于TCP_FIN_WAIT_2状态，先判断收到的数据包。如果不是FIN/ACK，直接drop掉。如果是，启动定时器。

当接收的报文确认了发送的报文，则更新窗口大小，如果报文有数据，接收数据。

当接收的报文标志位FIN为1，socket状态转变到TCP_CLOSE_WAIT状态后回复ACK报文。发送FIN报文，请求关闭这个连接，并将socket状态设置为TCP_LAST_ACK。

6、基于超时重传定时器的连接管理和数据传输

每个连接维护一个超时重传定时器，定时器管理涉及到：

- 当发送一个带数据或SYN|FIN的包，如果定时器是关闭的，则开启并设置时间为200ms
- 当ACK确认了部分数据，重启定时器，设置时间为200ms
- 当ACK确认了所有数据以及SYN|FIN，关闭定时器

触发定时器后：

- 重传第一个没有被对方连续确认的数据或SYN|FIN
- 定时器时间翻倍，记录该数据包的重传次数
- 当一个数据包重传3次，对方都没有确认，关闭该连接

```
void tcp_scan_timer_list()
{
	fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
}
```

void tcp_scan_timer_list()实现对timer_list的扫描，如果发现某个连接超过2MSL，则释放。

```
void tcp_set_timewait_timer(struct tcp_sock *tsk)
{
	fprintf(stdout, "TODO: implement %s please.\n",
__FUNCTION__);
}
```

void tcp_set_timewait_timer(struct tcp_sock *tsk)通过将定时器添加到timer_list中来设置tcp sock的timewait定时器

```
void tcp_set_retrans_timer(struct tcp_sock *tsk)
{
	fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
}
```

void tcp_set_retrans_timer(struct tcp_sock *tsk)函数通过将timer添加到timer_list中来设置tcp sock的retrans timer。

```
void tcp_unset_retrans_timer(struct tcp_sock *tsk)
{
	fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
}
```

void tcp_unset_retrans_timer(struct tcp_sock *tsk) 通过从timer_list中删除定时器来取消设置tcp sock的重新定时器。

四、结果与分析

1、TCP数据传输验证

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo.py)
- 在节点h1上执行TCP程序
  - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)，禁止协议栈的相应功能
  - 在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)

- 在节点h2上执行TCP程序
  - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)，禁止协议栈的相应功能 • 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)
  - Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat

- 使用md5sum比较两个文件是否完全相同
- 使用tcp_stack.py替换其中任意一端，对端都能正确收发数据

2、可靠传输验证

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)

- 在节点h1上执行TCP程序
    - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)

- 在节点h2上执行TCP程序
    - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)

- Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat
- 使用md5sum比较两个文件是否完全相同
- 使用tcp_stack.py替换两端任意一方，对端都能正确处理数据收发

3、TCP拥塞控制验证

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)
- 在节点h1上执行TCP程序
    - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)

- 在节点h2上执行TCP程序
    - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)

- Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat
- 使用md5sum比较两个文件是否完全相同
- 记录h2中每次cwnd调整的时间和相应值，呈现到二维坐标图中

五、小组分工