网络传输机制实验三报告

李昊宸

2017K8009929044

（一）TCP 可靠传输实现

一、实验内容

1. TCP server client 文件传输实验：

1）运行网络拓扑(tcp_topo_loss.py)

2）在节点 h1 上执行 TCP 程序

执行脚本(disable_tcp_rst.sh, disable_offloading.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

3）在节点 h2 上执行 TCP 程序

执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能

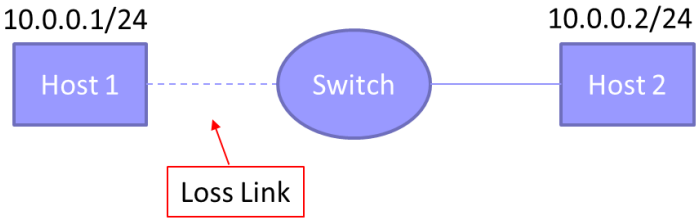
在 h2 上运行 TCP 协议栈的客户端模式：Client 发送文件 client-input.dat 给 server，server 将收到的数据存储到文件 server-output.dat

4）比较两个文件是否完全相同

二、实验流程

1. 搭建实验环境

arp.c arpcache.c icmp.c ip.c main.c packet.c rtable.c
rtable_internal.c
tcp_apps.c # 能够进行收发数据的 tcp sock apps
tcp.c : TCP 协议相关处理函数
tcp_in.c : TCP 接收相关函数
tcp_out.c : TCP 发送相关函数
tcp_sock.c : tcp_sock 操作相关函数
tcp_stack.py : python 应用实现，用于测试
tcp_timer.c : TCP 定时器
create_randfile.sh # 随机生成文件的脚本
tcp_topo_loss.py: 实现二节点的丢包率为 2%的拓扑



图一 丢包率为 2%的二节点网络拓扑

上周的实验已经实现 TCP 的稳定传输，这周需要补充在连接建立后进行数据的可靠传输。

TCP 的超时重传机制：为每个连接维护一个超时重传定时器。

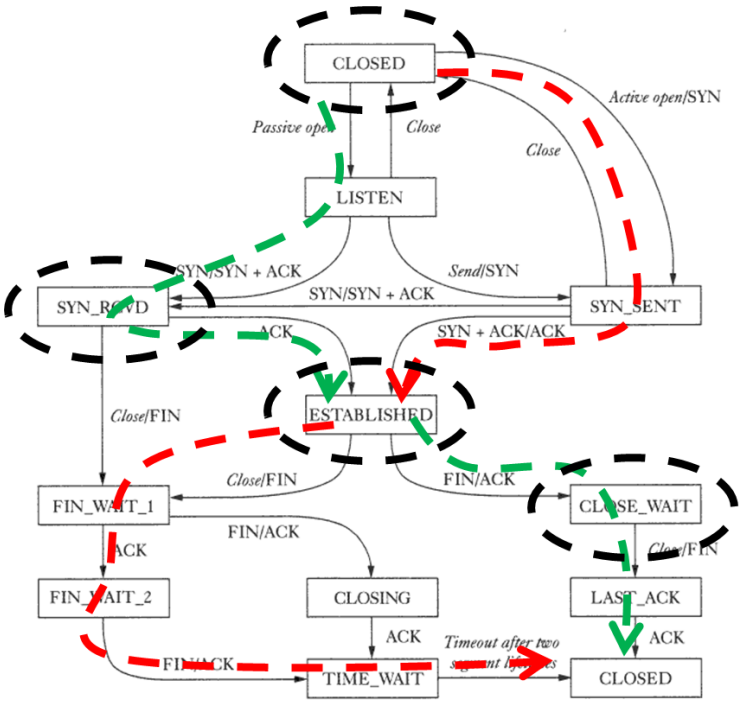
1）定时器管理

当发送一个带数据/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/SYN/FIN，关闭定时器。

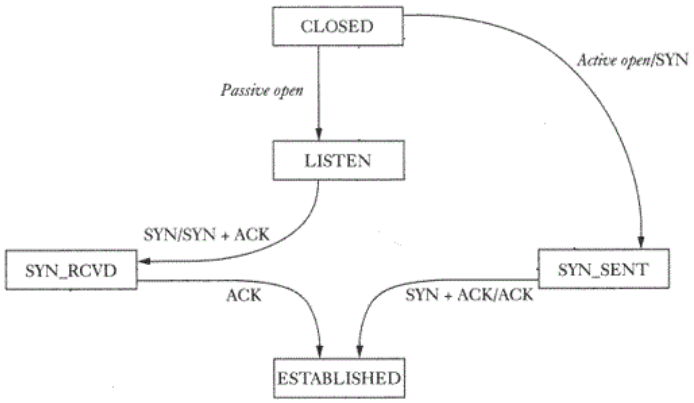
2）触发定时器后

重传第一个没有被对方连续确认的数据/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(RST)。

TCP 可能出现发送的包丢包的位置：



连接建立（有丢包）：



主动建立连接的场合：

1） **发送 SYN，该数据包被丢弃：**主动方状态为 SYN_SENT,被动方为 LISTEN，主动方超时后需要重新发送 SYN 包。

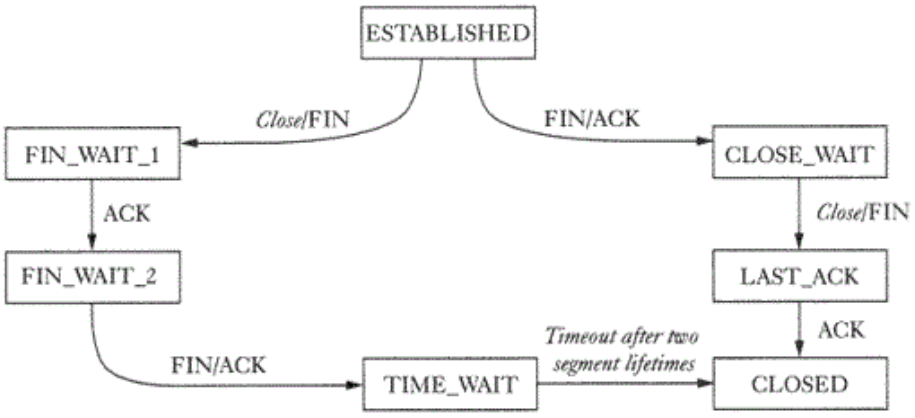
2） **被动方返回的 SYN|ACK 被丢弃：**主动方状态为 SYN_SENT,被动方为 SYN_RCVD，被动方超时后需要重新发送 SYN|ACK 包。

被动建立连接的场合：

1） **被动方返回的 SYN|ACK 被丢弃：**主动方状态为 SYN_SENT,被动方为 SYN_RCVD，被动方超时后需要重新发送 SYN|ACK 包。

2） **主动方返回的 ACK 被丢弃：**主动方状态为 ESTABLISHED,被动方为 SYN_RCVD，主动方超时后需要重新发送 ACK 包。

连接释放（有丢包）：



主动关闭连接的场合：

1） **发送 FIN|ACK，该数据包被丢弃：**主动方状态为 FIN_WAIT_1,被动方为 ESTABLISHED，主动方超时后需要重新发送 FIN|ACK 包。

2） **被动方返回的 ACK 被丢弃：**主动方状态为 FIN_WAIT_1,被动方为 CLOSE_WAIT，被动方超时后需要重新发送 ACK 包。

被动关闭连接的场合：

1） **被动方发送的 FIN|ACK 被丢弃：**主动方状态为 FIN_WAIT_2,被动方为 LAST_ACK，被动方超时后需要重新发送 FIN|ACK 包。

2） **主动方返回的 ACK 被丢弃：**主动方状态为 TIME_WAIT->CLOSED,被动方为 LAST_ACK，主动方超时后需要重新发送 ACK 包。

TCP 发送队列：

所有未确认的数据/SYN/FIN 包，在收到其对应的 ACK 之前，都要放在发送队列 snd_buffer（链表实现）中，以备后面可能的重传。

1）发送新的数据时

放到 snd_buffer 队尾，打开定时器（将对应 tsk 放入扫描队列中）。

2）收到新的 ACK

将 snd_buffer 中 seq_end <= ack 的数据包移除，并更新定时器。

3）重传定时器触发时

重传 snd_buffer 中第一个数据包，定时器数值翻倍。当重传次数超过三次时，断开连接。

4）切换状态时

建议关闭定时器（将对应 tsk 从扫描队列中删除），等到再次发送数据时重新开始定时器。

5）定时器扫描

每 10ms 扫描一次定时器队列。

TCP 接收队列：

数据接收方需要维护两个队列

1）已经连续收到的数据，放在 rcv_ring_buffer 中供 app 读取

2）收到不连续的数据，放到 rcv_ofo_buffer 队列（链表实现）中

接收方负责在收到数据包时回复相应 ACK，收到不连续的数据包时，按序放在 rcv_ofo_buffer 队列，如果队列中包含了连续数据，则将其移到 rcv_ring_buffer 中。

2. 启动脚本

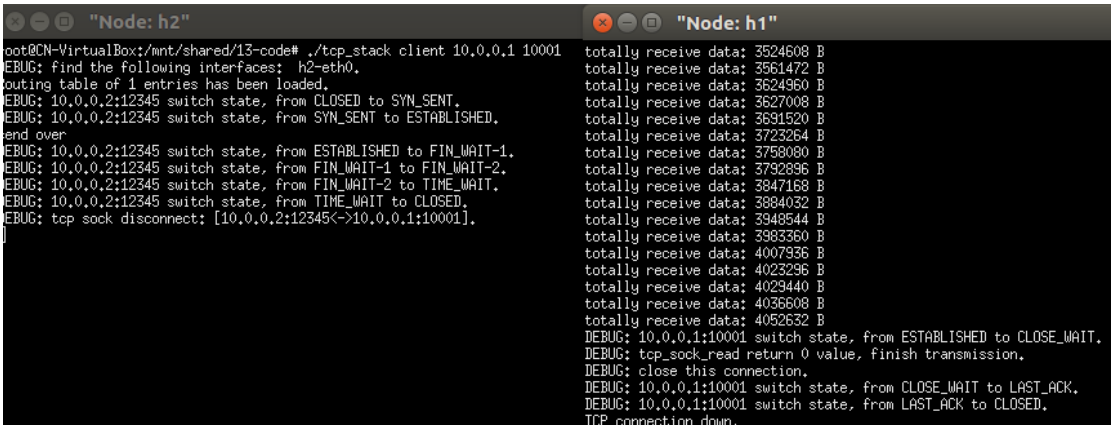
1) TCP server client 文件传输

```
make all
sudo python tcp_topo_loss.py
mininet> xterm h1 h2
h1# ./tcp_stack server 10001
h2# python tcp_stack.py client 10.0.0.1 10001
mininet> quit
```

注：上次实验已经验证状态转移的鲁棒性，故本次不再交叉验证（而且改 py 文件还蛮麻烦，就不改了 233）

三、实验结果及分析

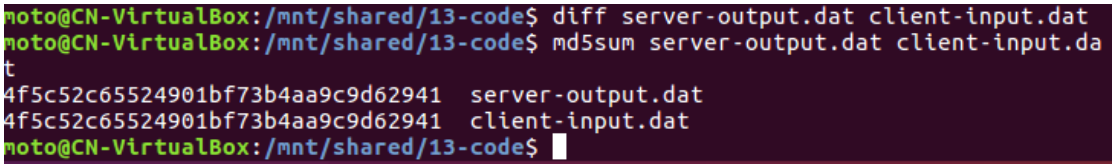
1. 实验结果



图二 TCP 可靠传输实验结果

可以看到，在三次握手连接建立完成后，client（h2）发送给 h1 的内容，在 h1 保存完之后，打印当前所收到的总比特数。

在数据传输完成后（达到了文件大小 4052632B），client（h2）发起关闭连接的请求，server（h1）响应之。整个状态变化过程与上一次实验相同。

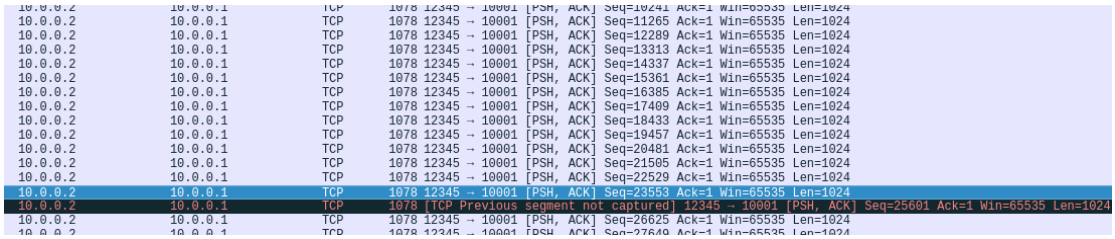


图三 TCP 可靠传输文件比较实验结果

调用 diff 命令比较 client_input.dat 和 server-output.dat，发现完全一致。

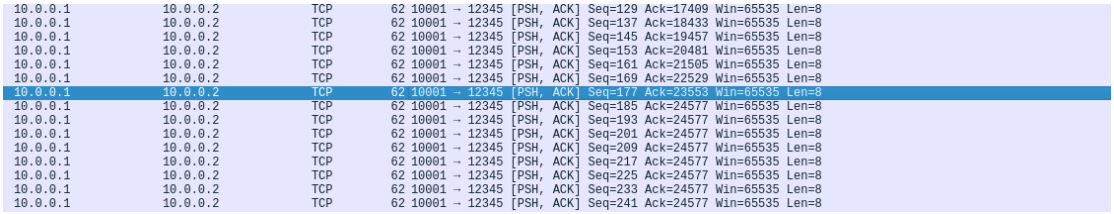
调用 md5sum 命令比较 client_input.dat 和 server-output.dat，发现完全一致。

说明实现了稳定传输。



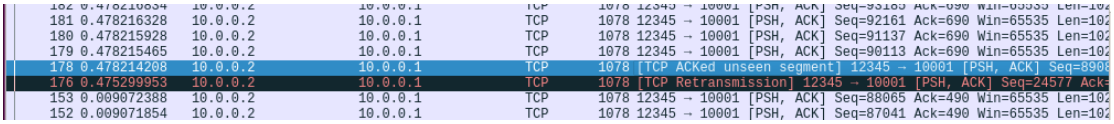
图四 wireshark 抓取丢失包

在 h1 节点的 eth0 端口进行抓包，发现在每个包大小为 1024B 的基础上，序列号为 23553 的包与序列号为 25601 的包中间的序列号为 24577 的包在传输过程中丢失。



图五 wireshark 抓取应答包

可以发现，在 h1 向 h2 发送的确认报文中，从 24577 开始序列号不再增长，说明接收方没有收到序列号 24577 的包，不再进行继续确认。



图六 wireshark 抓取丢失包

重传计时器超时的 h2，重新发送了序列号为 24577 的包，并被 h1 所接收。

以上表明了链路在具有丢包的情况下，传输仍然可以稳定进行，说明实现了稳定传输。

（二）实验代码详解

本次实验代码过长，不再赘述代码的内容。另外，本次实验较难，部分处理逻辑询问了认识的学长的处理模式（如对乱序到达队列 ofo 的处理）。

一、 tcp_sock.h:新增关于发送缓冲区与接收缓冲区的数据结构

（1）发送缓冲区：

```
struct tcp_send_buffer_block{
    struct list_head list;
    int len; //包的总长度
    char* packet;
};
```

每一个块表示在每一次调用 tcp_send_packet 时发送的包的缓存，len 记录 packet 的总长度。

```
struct tcp_send_buffer{
    struct list_head list;
    int size;
    pthread_mutex_t lock;
    pthread_t thread_retrans_timer;
} send_buffer;
```

每一个缓存块通过调用 list_add_tail 或 list_insert 按序列号插入到 list 上，timer 为重传计时器，lock 为访问数据区的互斥锁。

（2）乱序接收缓冲区：

```
struct tcp_ofo_block {
    struct list_head list;
    u32 seq;
    u32 len; //数据部分的总长度
    char* data;
};
```

每一个块表示在乱序收到发送的包时，将包解析缓存下来的数据，并且按照 seq 的顺序进行有序排列。seq 表示收到的包的序列号，len 表示收到的包的 数据段的长度，data 表示收到的数据。

二、 tcp_in.c:修改发送逻辑

（1）void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)：对发送逻辑进行修改

如果当前状态为 TCP_LISTEN:
在发送 SYN|ACK 的控制包前，调用 tcp_set_retrans_timer 启动重传计时器。

如果当前状态为 TCP_SYN_SENT：
由于之前发送出去 SYN 包，如果收到了新的包，并且该包为 SYN|ACK，那么说明之前发送的 SYN 包没有丢失，调用 send_buffer_ACK 处理发送缓存，调用 tcp_unset_retrans_timer 关闭定时器，状态切换到 ESTABLISHED。随后发送一个 ACK 控制包（注意，该控制包可以不用开启定时器，因为在下一次发送数据时，必将发送一个 ACK 包，就算该控制包丢失也是不可见的）。

如果当前状态为 TCP_SYN_RECV：
在收到 ACK 包后，说明之前发送的 SYN|ACK 包没有丢失，于是可以调用 send_buffer_ACK 处理发送缓存，调用 tcp_unset_retrans_timer 关闭重传计时器。

如果当前状态为 ESTABLISHED，并且收到的包不是 FIN：
如果收到一个单纯的确认 ACK 或者一个 ACK 仅带有“data_recv!”的数据（此处需要进行说明的是，通过对 tcp_send_control_packet 的修改，在发送控制包时，如果发送到包是一个单纯的 ACK 包或者为一个单纯的 RST 包，那么不对其进行缓存，因为不包含数据。除此之外发送的包都进行缓存。这导致一个问题，后去如果出现连续的单纯 ACK 包丢失的话，数据发送方就会陷入长期的等待，而接收方也没有新的报文需要发送，很可能陷入死锁。所以为避免这种情况，接收方在进行确认时，也需要向发送方发送一个带有数据的 ACK，数据规定为“data_recv!”，并没有实际意义，仅作为一个 ACK 的识别符，并且如果该 ACK 报文丢失的话，因为其带有数据，会被缓存并重传，避免了死锁。）说明该方为数据发送方，在更新发送窗口后，需要调用 send_buffer_ACK 处理发送缓存，调用 tcp_update_retrans_timer 更新重传计时器。

如果当前状态为 TCP_FIM_WAIT_1，并且收到的包不是 FIN：
说明发送的 FIN|ACK 到达，调用 send_buffer_ACK 处理发送缓存，调用 tcp_unset_retrans_timer 关闭重传计时器。

如果当前状态为 CLOSING，并且收到的包不是 FIN：
说明发送的 FIN|ACK 到达，调用 send_buffer_ACK 处理发送缓存，调用 tcp_unset_retrans_timer 关闭重传计时器。

如果当前状态为 TCP_LAST_ACK，并且收到的包不是 FIN：
说明发送的 FIN|ACK 到达，调用 send_buffer_ACK 处理发送缓存，调用 tcp_unset_retrans_timer 关闭重传计时器。

（2）void tcp_recv_data(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)：对接收逻辑进行修改

如果收到的 seq 小于 rcv_nxt，就丢弃该包。否则，调用 ofo_packet_enqueue 插入乱序到达缓冲区，调用 ofo_packet_dequeue 将乱序到达缓冲区的顺序部分读出到环形 buffer 中，并唤醒 tcp_sock_read 读取环形 buffer 中的数据（注意读完后会关闭上一次开启的重传计时器）。最后调用 tcp_set_retrans_timer 再次打开重

传计时器，发送 ACK（含数据 data_recv!）报文。

三、 tcp_sock.c:修改读写函数与缓冲区操作函数

（1）int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)：从环形 buf 中读取数据到应用

首先进入循环：检查环形 buf 是否为空。如果空，就调用 sleep_on(wait_recv)陷入睡眠，等待 tcp_handle_recv_data 唤醒。如果非空，就调用 read_ring_buffer 读取数据，调用 wake_up(wait_recv)唤醒 tcp_handle_recv_data，最后调用 tcp_unset_retrans_timer 关闭本次传输的计时器，返回读取数据的长度。

（2）int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)：将数据发送出去

发包之前，先调用 tcp_set_retrans_timer 打开重传计时器。

进入循环，循环结束的条件为剩余待发送长度等于 0：设置发送长度为剩余待发送长度与默认发送长度（1024）中的最小值。当 send_buffer 的大小加上本次发送长度大于发送窗口时，需要陷入睡眠等待（wait_send），直到小于等于发送窗口时，调用 tcp_send_data 将本次数据发送出去。

从循环出来后，意味着所有数据都已经发送出去。现在等待所有数据都被确认（wait_send）后，调用 tcp_unset_retrans_timer 关闭重传计时器。

（3）void tcp_sock_close(struct tcp_sock *tsk)：修改连接关闭函数

如果当前状态为 TCP_ESTABLISHED:
发送控制包 FIN|ACK 前，调用 tcp_set_retrans_timer 打开重传计时器。

如果当前状态为 TCP_CLOSE_WAIT：
发送控制包 FIN|ACK 前，调用 tcp_set_retrans_timer 打开重传计时器。

（4）void send_buffer_free()：三次重传后仍未收到 ACK，就将发送缓冲区清空，释放连接

（5）void send_buffer_RETRAN_HEAD(struct tcp_sock *tsk)：重传 send_buffer 中第一个数据包

首先 list_empty 查看链表是否为空。

如果非空，就调用 list_entry 访问链表的第一个乱序块，将乱序块装包，修改 ack 为 htonl（rcv_nxt），修改 tcp 和 ip 校验和，最后调用 ip_send_packet 发送。

（6）void send_buffer_ADD_TAIL(char* packet,int len)：发送新的数据时，将数据放到 send_buffer 队尾

首先 malloc 分配一个发送缓冲块。

随后修改 len 为数据包的大小，packet 为包的数据。

最后获取发送缓冲区互斥锁，将 send_buffer 的 size 增加 TCP 数据包的数据部分长度的大小，将缓冲块调用宏 list_add_tail 增添到链表尾部，释放互斥锁。

（7）void send_buffer_ACK(struct tcp_sock *tsk, u32 ack)：收到 ACK 后，将 send_buffer 中序列号小于 ack 的数据包删除

调用 list_for_each_entry_safe 安全遍历每个发送缓冲块：解析缓存报头，获得 seq。如果 seq 小于收到 ACK 报文的 ack 号，说明该缓存已经顺利被对端接收，可以删除，就获取互斥锁，将 send_buffer 的 size 减小 TCP 数据包的数据部分长度的大小，将缓冲块调用宏 list_delete_entry 从链表删除，释放缓冲块的空间，释放互斥锁。

（8）void ofo_packet_enqueue(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)：乱序到达的 packet 有序进入优先级缓存队列

新建一个乱序块,读取解析好的 tcp_context_block,设置乱序块的 seq 为 cb->seq，len 为 cb->pl_len，data 为收到包的数据段。调用 list_for_each_entry_safe 安全遍历乱序缓冲区，调用 less_than_32b 与当前遍历到的块进行序列号比较，如果新建的乱序块的 seq 更小，就调用 list_add_tail 将其插入。如果遍历结束仍没有插入，就将其加入链表尾部。

（9）int ofo_packet_dequeue(struct tcp_sock *tsk)：有序化的乱序到达缓冲区向环形 buffer 输出有序块

设置当前要接收的 seq 为 rcv_nxt，调用 list_for_each_entry_safe 遍历乱序缓冲区：

如果 seq 与当前遍历到块的 seq 相等，说明是连续块。当该块的 len 大于环形 buffer 的空闲区域时，就打印“sleep on buff_full”，陷入睡眠（wait_recv）。等到被 read 唤醒后，打印“wake up”，继续进行比较。直到 len 小于空闲区域后，调用 write_ring_buffer 将数据写入缓冲区,调用 wake_up(wait_recv)唤醒 tcp_sock_read，设置当前要接收的 seq 为 seq+len，设置 rcv_nxt 为 seq，调用 list_delete_entry 删除该节点，释放空间，继续遍历。

如果 seq 与当前遍历到块的 seq 不相等，说明不连续，结束扫描。

四、 tcp_timer.c:重传计时器

（1）void tcp_set_retrans_timer(struct tcp_sock *tsk)：启动

重传计时器

设置 typr 为 1（retrans），timeout 为 MIN_RETRANS_TIME（200ms），初始化重传次数为 0，调用 list_add_tail 将 restrans_timer->list 添加到重传链表上。最后将 ref_cnt 加 1。

（2）void tcp_update_retrans_timer(struct tcp_sock *tsk)：更新重传计时器

设置 typr 为 1（retrans），timeout 为 MIN_RETRANS_TIME（200ms），初始化重传次数为 0。

（3）void tcp_unset_retrans_timer(struct tcp_sock *tsk)：关闭重传计时器

调用 list_delete_entry 将 retrans_timer 从重传链表上删除，调用 free_tcp_sock 减引用。

（4）void *tcp_retrans_timer_thread(void *arg)：重传计时器线程

每 10ms 执行一次 tcp_scan_retrans_timer_list，没啥好说的。

（5）void tcp_scan_retrans_timer_list()：扫描函数

遍历重传链表：

timeout 减少 TCP_RETRANS_SCAN_INTERVAL（10ms），调用 retrans_timer_to_tcp_sock 找到该计时器所在的 tcp_socket。

如果超时，先检查重传次数：

如果重传次数超过 3 次，就将该计时器从链表上删除，如果 tcp_socket 为父 socket，就调用 tcp_bind_unhash 解绑定，调用 wait_exit(tsk->wait_connect)，wait_exit(tsk->wait_accept)，wait_exit(tsk->wait_recv)，wait_exit(tsk->wait_send)解除所有睡眠，设置状态为 TCP_CLOSED，释放 socket，调用 send_buffer_free 释放缓冲区。

如果重传次数没有超过三次，就将 retrans_times 加 1，timeout 设置为 10ms*2^retrans_times，调用 send_buffer_RETRAN_HEAD 进行重传。

如果没超时，就忽略。