

# 实验报告

## 网络传输机制实验三

### 一、实验内容

了解 TCP 的超时重传机制，了解有丢包情况下连接建立与断开的处理流程，设计实现发送队列和接收队列，实现 TCP 的可靠传输。

### 二、实验流程

1. 实现定时器相关功能。
2. 实现发送队列和接收队列。
3. 添加有丢包情况下的连接建立与断开处理和数据包的超时重传。
4. 在给定拓扑下验证可靠传输的正确性。

### 三、实验结果及分析

#### （一）TCP 功能实现思路

##### 1、定时器相关功能

`void tcp_set_retrans_timer(struct tcp_sock *tsk)`

函数，开启重发定时器。注意如果已经置起，则更新超时时间。

```
void tcp_set_retrans_timer(struct tcp_sock *tsk)
{
    if (tsk->retrans_timer.enable) {
        tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
        return;
    }
    tsk->retrans_timer.type = 1;
    tsk->retrans_timer.enable = 1;
    tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
    tsk->retrans_timer.retrans_time = 0;
    init_list_head(&tsk->retrans_timer.list);
    list_add_tail(&tsk->retrans_timer.list, &retrans_timer_list);
}
```

`void tcp_update_retrans_timer(struct tcp_sock *tsk)`

函数，更新重发定时器。用于已建立连接后传输数据时，如果发送队列为空，则将定时器关闭，并唤醒发送数据进程。

```
void tcp_update_retrans_timer(struct tcp_sock *tsk)
{
    if (list_empty(&tsk->send_buf) && tsk->retrans_timer.enable) {
        tsk->retrans_timer.enable = 0;
        list_delete_entry(&tsk->retrans_timer.list);
        wake_up(tsk->wait_send);
    }
}
```

```
void tcp_unset_retrans_timer(struct tcp_sock *tsk)
```

函数，关闭重发定时器。用于建立连接和断开连接过程，每个 SYN 或 FIN 包确认收到后关闭定时器。

```
void tcp_unset_retrans_timer(struct tcp_sock *tsk)
{
    if (!list_empty(&tsk->retrans_timer.list)) {
        tsk->retrans_timer.enable = 0;
        list_delete_entry(&tsk->retrans_timer.list);
        wake_up(tsk->wait_send);
    }
    else {
        log(ERROR, "unset an empty retrans timer\n");
    }
}
```

```
void tcp_scan_retrans_timer_list(void)
```

函数，扫描重发定时器队列，对超时的定时器进行处理：重发次数不超过上限的进行重发，否则直接断开连接，回收资源。

```
void *tcp_retrans_timer_thread(void *arg)
```

线程函数，负责每 10ms 扫描一次重发定时器队列。

## 2、发送队列和接收队列函数

发送队列项的数据结构如下：

```
typedef struct send_buffer_entry {
    struct list_head list;
    char *packet;
    int len;
} send_buffer_entry_t;
```

```
void tcp_add_send_buffer(struct tcp_sock *tsk, char *packet, int len)
```

函数，将发送的数据包添加到发送队列。

```
void tcp_delete_send_buffer(struct tcp_sock *tsk, u32 ack)
```

函数，根据对方返回的 ACK，将已经接收的数据包从队列中移除。

```
void tcp_delete_send_buffer(struct tcp_sock *tsk, u32 ack)
{
    send_buffer_entry_t *sb_entry, *sb_entry_q;
    list_for_each_entry_safe(sb_entry, sb_entry_q, &tsk->send_buf, list) {
        struct tcphdr *tcp = packet_to_tcp_hdr(sb_entry->packet);
        u32 seq = ntohl(tcp->seq);
        if (less_than_32b(seq, ack)) {
            list_delete_entry(&sb_entry->list);
            free(sb_entry->packet);
            free(sb_entry);
        }
    }
}
```

```
void tcp_retrans_send_buffer(struct tcp_sock *tsk)
```

函数，超时后重发当前队列中第一个数据包。

```
void tcp_retrans_send_buffer(struct tcp_sock *tsk)
{
    if (list_empty(&tsk->send_buf)) {
        log(ERROR, "no pakect to retrans\n");
        return;
    }
    send_buffer_entry_t *first_sb_entry = list_entry(tsk->send_buf.next, send_buffer_entry_t, list);
    char *packet = (char *)malloc(first_sb_entry->len);
    memcpy(packet, first_sb_entry->packet, first_sb_entry->len);
    struct iphdr *ip = packet_to_ip_hdr(packet);
    struct tcphdr *tcp = packet_to_tcp_hdr(packet);

    tcp->ack = htonl(tsk->rcv_nxt);
    tcp->checksum = tcp_checksum(ip, tcp);
    ip->checksum = ip_checksum(ip);
    int tcp_data_len = ntohs(ip->tot_len) - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE;

    tsk->snd_wnd -= tcp_data_len;
    log(DEBUG, "retrans seq: %u\n", ntohl(tcp->seq));

    ip_send_packet(packet, first_sb_entry->len);
}
```

接收队列项的数据结构如下：

```
typedef struct rcv_ofo_buf_entry {
    struct list_head list;
    char *data;
    int len;
    int seq;
} rcv_ofo_buf_entry_t;
```

```
void tcp_add_rcv_ofo_buffer(struct tcp_sock *tsk, struct tcp_cb *cb)
```

函数，将接收到的数据包按 seq 序列号放入接收队列中。注意需要判断序列号相同的情况，相同的包丢弃。虽然在进入该函数前有判断数据包有效的步骤，但是如果重发多个包，前一个包还在接收队列而没被实际接收更新时，就可能将后发的相同数据包判定为有效。因此这里必须添加判断序列号相同的处理。

```
void tcp_add_rcv_ofo_buffer(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    rcv_ofo_buf_entry_t *rcv_entry = (rcv_ofo_buf_entry_t *)malloc(sizeof(rcv_ofo_buf_entry_t));
    rcv_entry->seq = cb->seq;
    rcv_entry->len = cb->pl_len;
    rcv_entry->data = (char *)malloc(cb->pl_len);
    memcpy(rcv_entry->data, cb->payload, cb->pl_len);
    init_list_head(&rcv_entry->list);

    rcv_ofo_buf_entry_t *entry, *entry_q;
    list_for_each_entry_safe (entry, entry_q, &tsk->rcv_ofo_buf, list) {
        if (rcv_entry->seq == entry->seq) {
            return;
        }
        if (less_than_32b(rcv_entry->seq, entry->seq)) {
            list_add_tail(&rcv_entry->list, &entry->list);
            return;
        }
    }
    list_add_tail(&rcv_entry->list, &tsk->rcv_ofo_buf);
}
```

```
int tcp_move_rcv_ofo_buffer(struct tcp_sock *tsk)
```

函数，将接收队列中与当前 rcv\_nxt 相邻的连续包放入环形缓冲区，更新窗口。实现与之前的 handle\_rcv\_data 函数相同，不再赘述。

### 3、协议栈更新，连接建立、断开、数据的可靠传输

首先发送数据的函数需要进行修改，每次发送数据包时把数据包加入发送队列并设置定时器。

```
void tcp_send_packet(struct tcp_sock *tsk, char *packet, int len)
```

该函数发送实际有效的数据包，所有通过该函数的数据包都需要超时重发。

```
tsk->snd_nxt += tcp_data_len;
tsk->snd_wnd -= tcp_data_len;

tcp_add_send_buffer(tsk, packet, len);
tcp_set_retrans_timer(tsk);
```

```
void tcp_send_control_packet(struct tcp_sock *tsk, u8 flags)
```

该函数发送控制包，所有通过该函数的 FIN 包和 SYN 包都需要超时重发。

```
if ((flags != TCP_ACK) && !(flags & TCP_RST)) {  
    tcp_add_send_buffer(tsk, packet, pkt_size);  
    tcp_set_retrans_timer(tsk);  
}
```

然后是接收到数据包的处理，连接建立和断开过程的处理都差不多，都是上一个状态发出的包由下一状态负责验收，收到回应时清空发送队列，超时未收到回应则重发。纯 ACK 包不需要重发，也不添加到发送队列。以 SYN\_SENT 状态为例，收到 server 的 ACK | FIN 回应时清空发送队列，关闭定时器，更新到下一状态。

```
case TCP_SYN_SENT: {  
    if (tcp->flags & (TCP_ACK | TCP_SYN)) {  
        tcp_set_state(tsk, TCP_ESTABLISHED);  
        tsk->rcv_nxt = cb->seq + 1;  
        tsk->snd_una = cb->ack;  
        tcp_unset_retrans_timer(tsk);  
        tcp_delete_send_buffer(tsk, cb->ack);  
  
        tcp_set_state(tsk, TCP_ESTABLISHED);  
        wake_up(tsk->wait_connect);  
        tcp_send_control_packet(tsk, TCP_ACK);  
    }  
    return;  
}
```

对于无效的数据包进行丢弃，回复 ACK，无效数据包的来源是重发多次或者超过接收窗口。

```
if (!is_tcp_seq_valid(tsk, cb)) {  
    tcp_send_control_packet(tsk, TCP_ACK);  
    return;  
}
```

最后是 ESTABLISHED 状态收发数据的处理。如果是不带数据的 ACK 包，首先验证数据包序列号是否与希望收到的序列号相同，不同则丢弃。如果 ACK 更新，说明对方实际接收了数据包，将定时器重置。最后根据 ACK 更新发送队列和定时器。

```

else if (tcp->flags & TCP_ACK) {
    if (cb->pl_len == 0) {
        if (tsk->rcv_nxt != cb->seq) {
            return;
        }
        tsk->rcv_nxt = cb->seq;
        if (cb->ack > tsk->snd_una) {
            tsk->retrans_timer.retrans_time = 0;
            tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
        }
        tsk->snd_una = cb->ack;
        tcp_update_window_safe(tsk, cb);
        tcp_delete_send_buffer(tsk, cb->ack);
        tcp_update_retrans_timer(tsk);
    }
    else {
        if (!is_tcp_seq_valid(tsk, cb)) {
            return;
        }
        handle_rcv_data(tsk, cb);
    }
}
}

```

如果收到带数据的包，则交由 `handle_rcv_data` 函数处理。首先判断序列号，如果是希望收到的序列号之前的，说明已经收到过，将其丢弃。然后查看缓冲区是否已满，满了则先睡眠等待。未滿则将数据包放入接收队列，然后检查接收队列，将连接收到的数据包放入环形缓冲区。最后更新接收窗口，更新发送队列和定时器。

```

void handle_rcv_data(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    if (less_than_32b(cb->seq, tsk->rcv_nxt)) {
        tcp_send_control_packet(tsk, TCP_ACK);
        return;
    }

    while (ring_buffer_full(tsk->rcv_buf)) {
        sleep_on(tsk->wait_rcv);
    }

    tcp_add_rcv_ofo_buffer(tsk, cb);
    tcp_move_rcv_ofo_buffer(tsk);
    tsk->snd_una = greater_than_32b(cb->ack, tsk->snd_una) ? cb->ack : tsk->snd_una;

    tcp_delete_send_buffer(tsk, cb->ack);
    tcp_update_retrans_timer(tsk);

    tcp_send_control_packet(tsk, TCP_ACK);
}

```

然后还需要注意一点，接收到 `FIN` 包时的处理。原来是直接转为下一状态，这在我们的框架内没有问题，因为控制包和数据包是分开的。但是 `python` 的程序可能将最后一个数据包和 `FIN` 一起发送，如果不作处理会丢失最后一个数据包。因此我们要检查 `FIN` 包是否为空，如果有数据则调用 `handle_rcv_data` 函数处理。

```

tcp_set_state(tsk, TCP_CLOSE_WAIT);
if (cb->pl_len == 0) {
    tsk->rcv_nxt = cb->seq + 1;
    tsk->snd_una = cb->ack;
}
else {
    handle_rcv_data(tsk, cb);
    tsk->rcv_nxt += 1;
}
tcp_send_control_packet(tsk, TCP_ACK);
wake_up(tsk->wait_rcv);

```

## (二) 实验功能验证

### (1) 本实验 server 和 client

H1: 本实验 server

H2: 本实验 client



```

"Node: h1"
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 5840
DEBUG: write: 4160
DEBUG: write: 1460
DEBUG: write: 8540
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 7300
DEBUG: write: 2700
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 2632
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.

```

```

"Node: h2"
DEBUG: send: 10000, remain: 92632, total: (3960000/4052632)
DEBUG: retrans time: 1

DEBUG: retrans seq: 3951461

DEBUG: send: 10000, remain: 82632, total: (3970000/4052632)
DEBUG: send: 10000, remain: 72632, total: (3980000/4052632)
DEBUG: send: 10000, remain: 62632, total: (3990000/4052632)
DEBUG: send: 10000, remain: 52632, total: (4000000/4052632)
DEBUG: send: 10000, remain: 42632, total: (4010000/4052632)
DEBUG: send: 10000, remain: 32632, total: (4020000/4052632)
DEBUG: retrans time: 1

DEBUG: retrans seq: 4017301

DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.

```

MD5 验证

```

joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum client-input.dat
b78cf28e8d6d7e312feb816610287faf  client-input.dat
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum server-output.dat
b78cf28e8d6d7e312feb816610287faf  server-output.dat

```

H1、H2 能正确传输文件。

(2) 本实验 server 和标准 client

H1: 本实验 server

H2: 标准 client

```

"Node: h1"
DEBUG: write: 536
DEBUG: write: 1072
DEBUG: write: 536
DEBUG: write: 1072
DEBUG: write: 1072
DEBUG: write: 1072
DEBUG: write: 536
DEBUG: write: 1072
DEBUG: write: 1072
DEBUG: write: 1072
DEBUG: write: 536
DEBUG: write: 1072
DEBUG: write: 1072
DEBUG: write: 536
DEBUG: write: 2144
DEBUG: write: 1072
DEBUG: write: 2144
DEBUG: write: 320
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.

```





```
root@joker-linux:/mnt/hgfs/share/16-tcp_stack# ./create_randfile.sh
记录了3+0 的读入
记录了3+0 的写出
3000000 bytes (3.0 MB, 2.9 MiB) copied, 0.501952 s, 6.0 MB/s
root@joker-linux:/mnt/hgfs/share/16-tcp_stack# ./tcp_stack.py client 10.0.0.1 10001
root@joker-linux:/mnt/hgfs/share/16-tcp_stack#
```

## MD5 验证

```
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum client-input.dat
50d9d30ac2192cb73bbd0b670e353813  client-input.dat
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum server-output.dat
50d9d30ac2192cb73bbd0b670e353813  server-output.dat
```

H1、H2 能正确传输文件。

### (3) 标准 server 和本实验 client

H1: 标准 server

H2: 本实验 client



```
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
10000  
2920  
7080  
10000  
10000  
10000  
10000  
2632  
0  
root@joker-linux:/mnt/hgfs/share/16-tcp_stack#
```

```
"Node: h2"

DEBUG: retrans seq: 3930001

DEBUG: send: 10000, remain: 102632, total: (3950000/4052632)
DEBUG: send: 10000, remain: 92632, total: (3960000/4052632)
DEBUG: send: 10000, remain: 82632, total: (3970000/4052632)
DEBUG: send: 10000, remain: 72632, total: (3980000/4052632)
DEBUG: send: 10000, remain: 62632, total: (3990000/4052632)
DEBUG: send: 10000, remain: 52632, total: (4000000/4052632)
DEBUG: send: 10000, remain: 42632, total: (4010000/4052632)
DEBUG: retrans time: 1

DEBUG: retrans seq: 4002921

DEBUG: send: 10000, remain: 32632, total: (4020000/4052632)
DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

MD5 验证

```
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum client-input.dat
4d0a508696334549d12fc3ca75dbc96f  client-input.dat
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum server-output.dat
4d0a508696334549d12fc3ca75dbc96f  server-output.dat
```

H1、H2 能正确传输文件。

综上，本实验 server 和 client 能正确传输文件，可靠传输功能正确。

(4) 将丢包率提升到 10% 进行测试

H1: 本实验 server

H2: 本实验 client

```
"Node: h1"

DEBUG: write: 4380
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
DEBUG: write: 2700
DEBUG: write: 4380
DEBUG: write: 5620
DEBUG: write: 10000
DEBUG: write: 4380
DEBUG: write: 1460
DEBUG: write: 4160
DEBUG: write: 10000
DEBUG: write: 2920
DEBUG: write: 5840
DEBUG: write: 1240
DEBUG: write: 5840
DEBUG: write: 4160
DEBUG: write: 10000
DEBUG: write: 2632
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
```

```
"Node: h2"
DEBUG: retrans seq: 4005841
DEBUG: retrans time: 2
DEBUG: retrans seq: 4005841
DEBUG: send: 10000, remain: 32632, total: (4020000/4052632)
DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
DEBUG: retrans time: 1
DEBUG: retrans seq: 4028761
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
DEBUG: retrans time: 1
DEBUG: retrans seq: 4035841
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

MD5 验证

```
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum client-input.dat
cad5f78b5254ef137e15ef8a695006ba  client-input.dat
joker@joker-linux:/mnt/hgfs/share/16-tcp_stack$ md5sum server-output.dat
cad5f78b5254ef137e15ef8a695006ba  server-output.dat
```

H1、H2 能正确传输文件。

综上所述可以看出，本实验可靠传输功能的健壮性也比较好，在较高丢包率下也能正常工作。

## 四、实验总结

通过本次实验，我了解了 TCP 协议栈的可靠传输功能，了解了如何在有丢包情况下进行连接的建立、断开，同时也掌握了实现超时重传的方法，这让我对 TCP 协议有了更多的认识。