

网络传输机制实验二

学号： 2021K8009929010

姓名： 贾城昊

一、 实验题目：网络传输机制实验二

二、 实验任务：

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。

三、 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。

四、 实验过程

（一）TCP 重传定时器相关功能

1. 设置 TCP 重传定时器

`tcp_set_retrans_timer` 函数用于设置 TCP 重传定时器。首先检查重传定时器是否已启用，如果是，则直接更新超时时间即可。否则设置定时器的类型为 TCP 重传定时

器，启用定时器，设置超时时间，并设置重传次数为 0，最后将定时器添加到 TCP 重传定时器链表的末尾。

具体代码如下所示：

```
// set the retrans timer of a tcp sock, by adding the timer into timer_list
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 = TIMER_TYPE_RETRANS;
    tsk->retrans_timer.enable = 1;
    tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
    tsk->retrans_timer.retrans_time = 0;

    pthread_mutex_lock(&retrans_timer_list_lock);
    list_add_tail(&tsk->retrans_timer.list, &retrans_timer_list);
    pthread_mutex_unlock(&retrans_timer_list_lock);
}
```

2. 更新 TCP 重发定时器

tcp_update_retrans_timer 函数负责更新 TCP 重传定时器。用于已建立连接后传输数据时，如果发送队列为空，则禁用定时器，并从定时器列表中删除，并唤醒发送数据进程。

具体实现的代码如下：

```
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);
    }
}
```

3. 关闭 TCP 重传定时器

`tcp_unset_retrans_timer` 函数负责关闭重发定时器。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

具体的实现代码如下：

```
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");
    }
}
```

4. 扫描 TCP 重传定时器队列

`tcp_scan_retrans_timer_list` 函数负责扫描 TCP 重传定时器队列，减小定时器剩余时间，并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。另外，每次重发的间隔时间采用指数退避算法，具体来说，在该函数中下次重传的间隔时间是这次重传时间的 2 倍。

具体的实现代码如下：

```

void tcp_scan_retrans_timer_list(void)
{
    struct tcp_sock *tsk;
    struct tcp_timer *time_entry, *time_q;

    pthread_mutex_lock(&retrans_timer_list_lock);

    list_for_each_entry_safe(time_entry, time_q, &retrans_timer_list, list) {
        time_entry->timeout -= TCP_RETRANS_SCAN_INTERVAL;
        tsk = retrans_timer_to_tcp_sock(time_entry);
        if (time_entry->timeout <= 0) {
            if (time_entry->retrans_time >= MAX_RETRANS_NUM && tsk->state != TCP_CLOSED){
                list_delete_entry(&time_entry->list);
                if (!tsk->parent) {
                    tcp_unhash(tsk);
                }
                wait_exit(tsk->wait_connect);
                wait_exit(tsk->wait_accept);
                wait_exit(tsk->wait_rcv);
                wait_exit(tsk->wait_send);

                tcp_set_state(tsk, TCP_CLOSED);
                tcp_send_control_packet(tsk, TCP_RST);
            }
            else if (tsk->state != TCP_CLOSED) {
                time_entry->retrans_time += 1;
                log(DEBUG, "retrans time: %d\n", time_entry->retrans_time);

                time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL * (1 << time_entry->retrans_time);
                //time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL;
                tcp_retrans_send_buffer(tsk);
            }
        }
    }

    pthread_mutex_unlock(&retrans_timer_list_lock);
}

```

5. TCP 重传定时器对抗 i 额扫描线程

tcp_retrans_timer_thread 函数是一个线程函数，每 10ms 扫描一次重传定时器队列，实现 TCP 重传定时器的定时扫描。

具体的实现代码如下：

```

void *tcp_retrans_timer_thread(void *arg)
{
    init_list_head(&retrans_timer_list);
    while(1){
        usleep(TCP_RETRANS_SCAN_INTERVAL);
        tcp_scan_retrans_timer_list();
    }

    return NULL;
}

```

（二）发送队列的实现

1. 数据结构

每个发送队列项记录了 TCP 包的内容，包的长度等基本信息。具体来说，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;

```

2. 发送队列添加数据包

`tcp_send_buffer_add_packet` 函数负责将数据包添加到 TCP 发送队列。在访问发送队列时需要使用互斥锁，用来防止重传线程访问发送队列与收发包线程访问发送队列产生冲突。

具体实现的代码如下：

```
// Add a packet to the TCP send buffer
void tcp_send_buffer_add_packet(struct tcp_sock *tsk, char *packet, int len) {
    send_buffer_entry_t *send_buffer_entry = (send_buffer_entry_t *)malloc(sizeof(send_buffer_entry_t));
    memset(send_buffer_entry, 0, sizeof(send_buffer_entry_t));

    send_buffer_entry->packet = (char *)malloc(len);
    send_buffer_entry->len = len;
    memcpy(send_buffer_entry->packet, packet, len);

    init_list_head(&send_buffer_entry->list);

    pthread_mutex_lock(&tsk->send_buf_lock);
    list_add_tail(&send_buffer_entry->list, &tsk->send_buf);
    pthread_mutex_unlock(&tsk->send_buf_lock);
}
```

3. 更新发送队列

tcp_update_send_buffer 函数用于更新 TCP 发送队列，该函数基于收到的 ACK 包，遍历发送队列，将已经接收的数据包从队列中移(只需要判断包的序列号是否小于收到的 ACK 即可)。

同样的，在访问发送队列时需要使用互斥锁，用来防止重传线程访问发送队列与收发包线程访问发送队列产生冲突。

具体代码如下：

```

//Update the TCP send buffer based on the acknowledgment number
void tcp_update_send_buffer(struct tcp_sock *tsk, u32 ack) {
    send_buffer_entry_t *send_buffer_entry, *send_buffer_entry_q;
    pthread_mutex_lock(&tsk->send_buf_lock);

    list_for_each_entry_safe(send_buffer_entry, send_buffer_entry_q, &tsk->send_buf, list) {
        struct tcphdr *tcp = packet_to_tcp_hdr(send_buffer_entry->packet);
        u32 seq = ntohl(tcp->seq);

        // If the sequence number is less than the acknowledgment number, delete the entry
        if (less_than_32b(seq, ack)) {
            //log(DEBUG, "delete %d %d\n", seq, ack);
            list_delete_entry(&send_buffer_entry->list);
            free(send_buffer_entry->packet);
            free(send_buffer_entry);
        }
    }
    pthread_mutex_unlock(&tsk->send_buf_lock);
}

```

4. 发送队列数据包的超时重传

tcp_retrans_send_buffer 函数负责超时后重发当前队列中第一个数据包。在获取发送队列中的第一个数据包后，复制数据包并更新 TCP 序列号、确认号、校验和等信息，计算 TCP 数据长度并更新 TCP 发送窗口，最后进行发送。

具体代码如下：

```

// Retransmit the first packet in the TCP send buffer when ack time exceed
int tcp_retrans_send_buffer(struct tcp_sock *tsk) {
    pthread_mutex_lock(&tsk->send_buf_lock);

    if (list_empty(&tsk->send_buf)) {
        log(ERROR, "no packet to retrans\n");
        pthread_mutex_unlock(&tsk->send_buf_lock);
        return 0;
    }

    // Retrieve the first send buffer entry
    send_buffer_entry_t *first_send_buffer_entry = list_entry(tsk->send_buf.next, send_buffer_entry_t, list);

    char *packet = (char *)malloc(first_send_buffer_entry->len);

    // Copy the packet data and update TCP sequence and acknowledgment numbers
    memcpy(packet, first_send_buffer_entry->packet, first_send_buffer_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);

    // Calculate TCP data length and update TCP send window
    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));
    pthread_mutex_unlock(&tsk->send_buf_lock);

    // Send the packet
    ip_send_packet(packet, first_send_buffer_entry->len);

    return 1;
}

```

(三) 接收队列的实现

1. 数据结构

接收队列可以存储乱序收到的TCP包,每个接收队列项记录了收到的TCP包的内容,长度以及序列号和结束序列号。具体来说,接收队列项的数据结构如下:

```

typedef struct recv_ofo_buf_entry {
    struct list_head list;
    char *data;
    int len;
    u32 seq;
    u32 seq_end;
} recv_ofo_buf_entry_t;

```


2. 接收队列添加数据包

`tcp_rcv_ofo_buffer_add_packet` 将接收到的数据包放入接收队列中。在这个函数中，数据包的插入是按照 seq 顺序的。

在插入过程中，如果发现序列号相同的情况，则需要把相同的包丢弃。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。

具体代码如下所示：

```
// Add an packet to the TCP receive buffer
int tcp_rcv_ofo_buffer_add_packet(struct tcp_sock *tsk, struct tcp_cb *cb) {
    if (cb->pl_len <= 0) {
        return 0;
    }

    rcv_ofo_buf_entry_t *rcv_ofo_entry = (rcv_ofo_buf_entry_t *)malloc(sizeof(rcv_ofo_buf_entry_t));
    rcv_ofo_entry->seq = cb->seq;
    rcv_ofo_entry->seq_end = cb->seq_end;
    rcv_ofo_entry->len = cb->pl_len;
    rcv_ofo_entry->data = (char *)malloc(cb->pl_len);
    memcpy(rcv_ofo_entry->data, cb->payload, cb->pl_len);

    init_list_head(&rcv_ofo_entry->list);

    // insert the new entry at the correct position
    rcv_ofo_buf_entry_t *entry, *entry_q;
    list_for_each_entry_safe(entry, entry_q, &tsk->rcv_ofo_buf, list) {
        if (rcv_ofo_entry->seq == entry->seq) {
            return 1; // same seq, do not add
        }
        if (less_than_32b(rcv_ofo_entry->seq, entry->seq)) {
            list_add_tail(&rcv_ofo_entry->list, &entry->list);
            return 1;
        }
    }
    list_add_tail(&rcv_ofo_entry->list, &tsk->rcv_ofo_buf);

    return 1;
}
```

3. 移动接收队列的数据包

`tcp_move_rcv_ofo_buffer` 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。具体来说，遍历整个接收队列，发现对应项的序列号与 `rcv_nxt` 相等则写进 `ring_buffer` 中（需要互斥访问，具体原因可以参见上次实验实验报告对于 `ring_buffer` 读写操作的解释），并唤醒 `wait_rcv`，然后更新 `rcv_nxt`，继续遍历。这样就能实现把跟当前 `rcv_nxt` 连续的数据包添加到环形缓存区中，并正确更新 `rcv_nxt`。

具体代码如下所示：

```
// Move packets from TCP receive buffer to ring buffer
int tcp_move_rcv_ofo_buffer(struct tcp_sock *tsk) {
    rcv_ofo_buf_entry_t *entry, *entry_q;

    list_for_each_entry_safe(entry, entry_q, &tsk->rcv_ofo_buf, list) {
        if (tsk->rcv_nxt == entry->seq) {
            // Wait until there is enough space in the receive buffer
            while (ring_buffer_free(tsk->rcv_buf) < entry->len) {
                sleep_on(tsk->wait_rcv);
            }

            pthread_mutex_lock(&tsk->rcv_buf_lock);
            write_ring_buffer(tsk->rcv_buf, entry->data, entry->len);
            tsk->rcv_wnd -= entry->len;
            pthread_mutex_unlock(&tsk->rcv_buf_lock);
            wake_up(tsk->wait_rcv);

            // Update seq and free memory
            tsk->rcv_nxt = entry->seq_end;
            list_delete_entry(&entry->list);
            free(entry->data);
            free(entry);
            //log(DEBUG, "find : %d", entry->seq_end);
        }
        else if (less_than_32b(tsk->rcv_nxt, entry->seq)) {
            continue; //the next expected sequence number is not reached yet
        }
        else {
            log(ERROR, "rcv_nxt is more than seq, rcv_nxt: %d, seq: %d\n", tsk->rcv_nxt, entry->seq);
            return 0;
        }
    }

    return 1;
}
```

（四）协议栈函数的更新：实现可靠传输

1. 数据包的发送

tcp_send_packet 函数负责发送实际有效的数据包。在本次实验中，为了实现可靠传输，所有通过该函数的数据包都需要添加到发送队列里面，并设置超时重发定时器。

具体代码如下所示

```
void tcp_send_packet(struct tcp_sock *tsk, char *packet, int len)
{
    struct iphdr *ip = packet_to_ip_hdr(packet);
    struct tcphdr *tcp = (struct tcphdr *)((char *)ip + IP_BASE_HDR_SIZE);

    int ip_tot_len = len - ETHER_HDR_SIZE;
    int tcp_data_len = ip_tot_len - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE;

    u32 saddr = tsk->sk_sip;
    u32 daddr = tsk->sk_dip;
    u16 sport = tsk->sk_sport;
    u16 dport = tsk->sk_dport;

    u32 seq = tsk->snd_nxt;
    u32 ack = tsk->rcv_nxt;
    u16 rwnd = tsk->rcv_wnd;

    tcp_init_hdr(tcp, sport, dport, seq, ack, TCP_PSH|TCP_ACK, rwnd);
    ip_init_hdr(ip, saddr, daddr, ip_tot_len, IPPROTO_TCP);

    tcp->checksum = tcp_checksum(ip, tcp);

    ip->checksum = ip_checksum(ip);

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

    tcp_send_buffer_add_packet(tsk, packet, len);
    tcp_set_retrans_timer(tsk);

    ip_send_packet(packet, len);
}
```

2. 控制包的发送

tcp_send_control_packet 函数负责发送控制包。在本次实验中，为了实现可靠传输，所有通过该函数的 FIN 包和 SYN 包都需要设置超时重发定时器。

具体代码如下所示：

```

void tcp_send_control_packet(struct tcp_sock *tsk, u8 flags)
{
    int pkt_size = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;
    char *packet = malloc(pkt_size);
    if (!packet) {
        log(ERROR, "malloc tcp control packet failed.");
        return ;
    }

    struct iphdr *ip = packet_to_ip_hdr(packet);
    struct tcphdr *tcp = (struct tcphdr *)((char *)ip + IP_BASE_HDR_SIZE);

    u16 tot_len = IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;

    ip_init_hdr(ip, tsk->sk_sip, tsk->sk_dip, tot_len, IPPROTO_TCP);
    tcp_init_hdr(tcp, tsk->sk_sport, tsk->sk_dport, tsk->snd_nxt, \
        tsk->rcv_nxt, flags, tsk->rcv_wnd);

    tcp->checksum = tcp_checksum(ip, tcp);

    if (flags & (TCP_SYN|TCP_FIN))
        tsk->snd_nxt += 1;

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

    ip_send_packet(packet, pkt_size);
}

```

3. TCP 数据包与连接管理

tcp_process 函数负责处理 TCP 数据包和连接管理。在本次实验中，为了实现可靠传输，需要进行以下修改：

首先是在连接建立过程中，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 SYN | ACK 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 (cb->flags == (TCP_SYN | TCP_ACK)) {
        tsk->rcv_nxt = cb->seq_end;

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

        tcp_unset_retrans_timer(tsk);
        tcp_update_send_buffer(tsk, cb->ack);

        tcp_set_state(tsk, TCP_ESTABLISHED);

        // send ACK;
        tcp_send_control_packet(tsk, TCP_ACK);

        wake_up(tsk->wait_connect);
    }
}

```

SYN_RECV 状态的变化也同理，收到 ACK 后清空发送队列，关闭定时器，更新到

下一状态：

```

case(TCP_SYN_RECV):{
    if (cb->flags == TCP_ACK) {
        if (!is_tcp_seq_valid(tsk, cb)) {
            return;
        }
        tsk->rcv_nxt = cb->seq_end;

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

        if (tsk->parent) {
            if (tcp_sock_accept_queue_full(tsk->parent)) {
                tcp_set_state(tsk, TCP_CLOSED);
                // send RST
                tcp_send_control_packet(tsk, TCP_RST);

                tcp_unhash(tsk);
                tcp_bind_unhash(tsk);

                // remove from listen list
                list_delete_entry(&tsk->list);
                free_tcp_sock(tsk);
                log(DEBUG, "tcp_sock accept queue is full, so the tsk should be freed.");
            }
            else {
                tcp_set_state(tsk, TCP_ESTABLISHED);
                tcp_sock_accept_enqueue(tsk);

                tcp_unset_retrans_timer(tsk);
                tcp_update_send_buffer(tsk, cb->ack);

                // wake up user process for accept
                wake_up(tsk->parent->wait_accept);
            }
        }
    }
}

```

在关闭建立的过程中，也与上面说的类似，收到 ACK 后清空发送队列，关闭定时器，更新到下一状态，具体的相关实现代码如下：

```
case(TCP_FIN_WAIT_1):{
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }

    // do something but not this stage
    tsk->rcv_nxt = cb->seq_end;

    if (cb->flags & TCP_ACK) {
        tcp_update_send_buffer(tsk, cb->ack);
        tcp_unset_retrans_timer(tsk);

        tcp_update_window_safe(tsk, cb);
        tsk->snd_una = cb->ack;
    }

    if ((cb->flags & TCP_FIN) && (cb->flags & TCP_ACK) && tsk->rcv_nxt == cb->seq_end) {
        tcp_set_state(tsk, TCP_CLOSING);
        tsk->rcv_nxt = cb->seq_end;
        tsk->snd_una = cb->ack;
    }
}

case(TCP_LAST_ACK):{
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }

    tsk->rcv_nxt = cb->seq_end;

    if (cb->flags & TCP_ACK) {
        tcp_update_window_safe(tsk, cb);
        tsk->snd_una = cb->ack;
    }

    if ((cb->flags & TCP_ACK) && tsk->snd_nxt == tsk->snd_una) {
        tcp_update_send_buffer(tsk, cb->ack);
        tcp_unset_retrans_timer(tsk);
        tcp_set_state(tsk, TCP_CLOSING);
        tsk->rcv_nxt = cb->seq;
        tsk->snd_una = cb->ack;

        tcp_set_state(tsk, TCP_CLOSING);

        // release the sock
        tcp_unhash(tsk);
        tcp_bind_unhash(tsk);

        free_tcp_sock(tsk);
    }
    break;
}
```

最后是 ESTABLISH 状态，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，则不做处理，直接回复 ACK 即可。

```

if (less_than_32b(cb->seq, tsk->rcv_nxt)) {
    tcp_send_control_packet(tsk, TCP_ACK);
    //log(DEBUG, "received packet had been received before, drop it.");
    return;
}

```

然后，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。具体相关代码如下：

```

else {
    if (cb->pl_len != 0) {
        handle_tcp_rcv_data(tsk, cb);
        //tcp_send_control_packet(tsk, TCP_ACK);
    }
    else{
        tsk->rcv_nxt = cb->seq_end;
        if (cb->ack > tsk->snd_una) {
            tsk->retrans_timer.retrans_time = 0;
            tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
        }
        //printf("%d %d\n", tsk->snd_una, cb->ack);
        tsk->snd_una = cb->ack;
        tcp_update_window_safe(tsk, cb);
        tcp_update_send_buffer(tsk, cb->ack);
        tcp_update_retrans_timer(tsk);
    }
}
break;

```

如果收到带数据的包，则交由 handle_tcp_rcv_data 函数处理。首先判断数据包长度是否合法，不合法则将其丢弃。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，如果有序列号与 rcv_nxt 相同的数据包，则把收到的与其序列号连续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。具体相关代码如下：

```

// handle the recv data from TCP packet
int handle_tcp_recv_data(struct tcp_sock *tsk, struct tcp_cb * cb) {
    if (cb->pl_len <= 0) {
        return 0;
    }

    pthread_mutex_lock(&tsk->rcv_buf_lock);

    while (ring_buffer_full(tsk->rcv_buf)) {
        pthread_mutex_unlock(&tsk->rcv_buf_lock);
        sleep_on(tsk->wait_recv);
    }

    tcp_recv_ofo_buffer_add_packet(tsk, cb);
    pthread_mutex_unlock(&tsk->rcv_buf_lock);

    tcp_move_recv_ofo_buffer(tsk);
    pthread_mutex_lock(&tsk->rcv_buf_lock);

    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);

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

    //fprintf(stderr, "%d %d\n", cb->seq, tsk->rcv_nxt);
    tcp_send_control_packet(tsk, TCP_ACK);

    wake_up(tsk->wait_recv);

    pthread_mutex_unlock(&tsk->rcv_buf_lock);

    return 1;
}

```

五、实验结果与分析

1. 编写 Python 脚本

参考 tcp_stack_trans.py 编写 python 文件 tcp_stack_file.py, 使之能实现文件的传输, 具体的代码如下:

(注: 本次实验的 python 脚本与上次一样)


```

def server(port, filename):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    s.bind(('0.0.0.0', int(port)))
    s.listen(3)

    cs, addr = s.accept()
    print addr

    with open(filename, 'wb') as f:
        while True:
            data = cs.recv(1024)
            if data:
                f.write(data)
            else:
                break

    s.close()

def client(ip, port, filename):
    s = socket.socket()
    s.connect((ip, int(port)))

    f = open('client-input.dat', 'r')
    file_str = f.read()
    length = len(file_str)
    i = 0

    while length > 0:
        send_len = min(length, 100000)
        s.send(file_str[i: i+send_len])
        sleep(0.1)
        print("send:{0}, remain:{1}, total: {2}/{3}".format(i, length, i, i+length))
        length -= send_len
        i += send_len

    f.close()
    s.close()

if __name__ == '__main__':
    if sys.argv[1] == 'server':
        server(sys.argv[2], "server-output.dat")
    elif sys.argv[1] == 'client':
        client(sys.argv[2], sys.argv[3], "client-input.dat")

```

2. 本实验 Server 与本实验 Client

H1: 本实验 server

H2: 本实验 client

运行结果如下图所示:

```

"Node: h1"
root@Computer:~/workspace/Network/lab14/14-tcp_stack3# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:12345 from process to listen_queue
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: peer closed.
used time: 10 s
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:12345, state CLOSE_WAIT
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.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:12345.

```

```

"Node: h2"
DEBUG: sent 3944728 Bytes
DEBUG: retrans time: 1

DEBUG: retrans seq: 3911981

DEBUG: sent 3964752 Bytes
DEBUG: sent 3984776 Bytes
DEBUG: sent 4004800 Bytes
DEBUG: sent 4024824 Bytes
DEBUG: retrans time: 1

DEBUG: retrans seq: 3994997

DEBUG: sent 4044848 Bytes
DEBUG: sent 4052632 Bytes
DEBUG: the file has been sent completely.
DEBUG: close sock 10.0.0.2:12345 <-> 10.0.0.1:10001, state ESTABLISHED
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: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: Free 10.0.0.2:12345 <-> 10.0.0.1:10001.

```

可以看到，虽然传输过程中出现了丢包的现象，但能正常进行重传，完成传输过程。并且传输只花费了 10s，在预计时间范围内。

MD5 验证结果如下：

```

sai@Computer: ~/workspace/Network/lab14/14-tcp_stack3
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum server-output.dat
05f7c5290651bce2fa0369f1921b3ccb  server-output.dat
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb  client-input.dat
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$

```

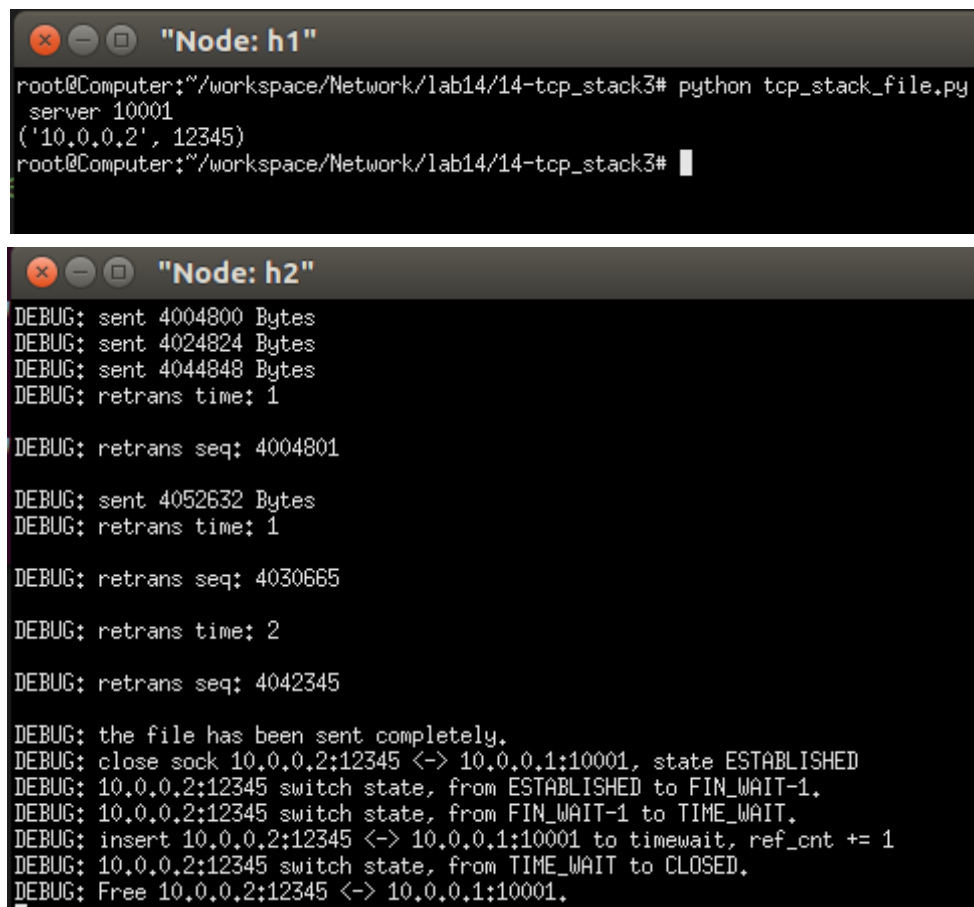
可以看出，最后服务器端接收到的文件与传输的文件一致。

3. 标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client

运行结果如下图所示：



```
"Node: h1"
root@Computer:~/workspace/Network/lab14/14-tcp_stack3# python tcp_stack_file.py
server 10001
('10.0.0.2', 12345)
root@Computer:~/workspace/Network/lab14/14-tcp_stack3#

"Node: h2"
DEBUG: sent 4004800 Bytes
DEBUG: sent 4024824 Bytes
DEBUG: sent 4044848 Bytes
DEBUG: retrans time: 1

DEBUG: retrans seq: 4004801

DEBUG: sent 4052632 Bytes
DEBUG: retrans time: 1

DEBUG: retrans seq: 4030665

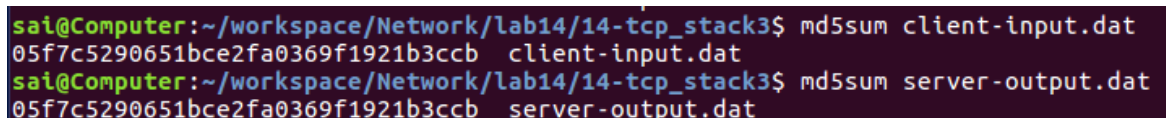
DEBUG: retrans time: 2

DEBUG: retrans seq: 4042345

DEBUG: the file has been sent completely.
DEBUG: close sock 10.0.0.2:12345 <-> 10.0.0.1:10001, state ESTABLISHED
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 TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: Free 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

可以看出，虽然传输过程中出现了丢包的现象，但能正常进行重传，完成传输过程。

MD5 验证结果如下：



```
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb client-input.dat
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum server-output.dat
05f7c5290651bce2fa0369f1921b3ccb server-output.dat
```

可以看出，最后服务器端接收到的文件与传输的文件一致。

4. 本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab14/14-tcp_stack3# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:54428 from process to listen_queue
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: peer closed.
used time: 14 s
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:54428, state CLOSE_WAIT
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.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:54428.
```

```
"Node: h2"
send:1800000, remain:2252632, total: 1800000/4052632
send:1900000, remain:2152632, total: 1900000/4052632
send:2000000, remain:2052632, total: 2000000/4052632
send:2100000, remain:1952632, total: 2100000/4052632
send:2200000, remain:1852632, total: 2200000/4052632
send:2300000, remain:1752632, total: 2300000/4052632
send:2400000, remain:1652632, total: 2400000/4052632
send:2500000, remain:1552632, total: 2500000/4052632
send:2600000, remain:1452632, total: 2600000/4052632
send:2700000, remain:1352632, total: 2700000/4052632
send:2800000, remain:1252632, total: 2800000/4052632
send:2900000, remain:1152632, total: 2900000/4052632
send:3000000, remain:1052632, total: 3000000/4052632
send:3100000, remain:952632, total: 3100000/4052632
send:3200000, remain:852632, total: 3200000/4052632
send:3300000, remain:752632, total: 3300000/4052632
send:3400000, remain:652632, total: 3400000/4052632
send:3500000, remain:552632, total: 3500000/4052632
send:3600000, remain:452632, total: 3600000/4052632
send:3700000, remain:352632, total: 3700000/4052632
send:3800000, remain:252632, total: 3800000/4052632
send:3900000, remain:152632, total: 3900000/4052632
send:4000000, remain:52632, total: 4000000/4052632
root@Computer:~/workspace/Network/lab14/14-tcp_stack3#
```

可以看出 H1 和 H2 能正常完成传输过程。并且传输只花费 14s，也在预期时间范围内。

MD5 验证结果如下：

```
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum server-output.dat
05f7c5290651bce2fa0369f1921b3ccb  server-output.dat
sai@Computer:~/workspace/Network/lab14/14-tcp_stack3$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb  client-input.dat
```

可以看出，最后服务器端接收到的文件与传输的文件一致。

综上，本实验 server 和 client 能正确收发文件，功能正确。

六、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，为之后的实验奠定了基础。