Report 14 — December 16

Lecturer: Wu Qinghua *Completed by: 2022K8009929010 Zhang Jiawei*

14.1 实验内容

1. 丢包恢复

   - 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` 和 `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` 和 `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。

2. 拥塞控制

   - 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` 和 `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` 和 `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。

14.2 实验过程

14.2.1 丢包恢复

1. 重传定时器操作

   本次实验中，我们需要实现定时器的设置、更新、关闭、扫描操作。

   设置操作较为简单，先检查是否有定时器，如果没有则直接设置定时器类型为重传定时器，启用并设置超时时间，设置重传次数为 0；如果已有定时器，则更新超时时间。最后把定时器加入定时器链表。

   更新操作也较为简单，若已建立的连接发送队列为空，则关闭并删除定时器，唤醒发送数据进程。

   关闭操作也较为简单，与更新操作类似，只是判断条件改为定时器队列不为空。

   扫描操作则需要遍历定时器链表，减少每个定时器的剩余时间，若剩余时间小于等于 0，即超时，则进行处理：重传次数未达上限则重传数据包，重传次数达到上限则直接断开连接，释放资源。扫描操作通过一个线程每 10 秒进行一次扫描。

   代码如下：

```
// set the restrans 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);
}

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)
```

```
{
    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)
{
    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 = retranstimer_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_recv);
                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;
                tcp_retrans_send_buffer(tsk);
            }
        }
    }
```

```c
    pthread_mutex_unlock(&retrans_timer_list_lock);
}

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;
}
```

2. 发送队列

   对于发送队列，我们需要实现数据包的添加、更新、重传操作，都比较简单。

   添加操作只需将数据包加入发送队列即可。注意这里需要使用互斥锁保护发送队列。

   更新操作是指遍历队列，将队列中序列号小于收到的 ACK 的数据包删除。这里同样需要使用互斥锁保护发送队列。

   重传操作是指超时未收到 ACK 时，将队列中第一个的数据包重传。我将队列中第一个数据包的 TCP 序列号、确认号等信息更新之后，再计算出数据长度和发送窗口大小，将数据包发送出去。

   代码如下：

```c
// 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);

    list_add_tail(&send_buffer_entry->list, &tsk->send_buf);
}

//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;

        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)) {
                list_delete_entry(&send_buffer_entry->list);
                free(send_buffer_entry->packet);
                free(send_buffer_entry);
            }
        }
    }

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

        // Send the packet
        ip_send_packet(packet, first_send_buffer_entry->len);
        return 1;
    }
```

3. 接收队列

对于接收队列，我们需要实现数据包的添加、移动操作，稍微繁琐。

添加操作要将接收到的数据包按照 seq 顺序插入接收队列中，若出现重复数据包则直接丢弃，然后将有效的数据包插入队列中。

移动操作首先遍历接收队列，找到与当前 `rcv_nxt` 匹配的数据包，然后写入环形缓冲区，唤醒接收进程，更新 `rcv_nxt`。注意环形缓冲区需要使用互斥锁保护。

代码如下：

```
    // Add an packet to the TCP receive buffer
    int tcp_recv_ofo_buffer_add_packet(struct tcp_sock *tsk, struct tcp_cb *cb) {
        if (cb->pl_len <= 0)
            return 0;
        recv_ofo_buf_entry_t *recv_ofo_entry = (recv_ofo_buf_entry_t
             *)malloc(sizeof(recv_ofo_buf_entry_t));
        recv_ofo_entry->seq = cb->seq;
        recv_ofo_entry->seq_end = cb->seq_end;
        recv_ofo_entry->len = cb->pl_len;
        recv_ofo_entry->data = (char *)malloc(cb->pl_len);
        memcpy(recv_ofo_entry->data, cb->payload, cb->pl_len);

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

```c
// Move packets from TCP receive buffer to ring buffer
int tcp_move_recv_ofo_buffer(struct tcp_sock *tsk) {
    recv_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_recv);

            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_recv);

            // Update seq and free memory
            tsk->rcv_nxt = entry->seq_end;
            list_delete_entry(&entry->list);
            free(entry->data);
            free(entry);
        }
        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;
}
```

4. **TCP 核心函数更新**

为了实现可靠传输,tcp_send_packet 函数需要在发送数据包时将数据包加入发送队列,设置重传定时器:

```c
// send a tcp packet
//
// Given that the payload of the tcp packet has been filled, initialize the tcp
// header and ip header (remember to set the checksum in both header), and emit
// the packet by calling ip_send_packet.
void tcp_send_packet(struct tcp_sock *tsk, char *packet, int len)
{
```

```c
    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);
}
```

同样,为了实现可靠传输,tcp_send_control_packet 函数需要在发送控制包时设置重传定时器:

```c
// send a tcp control packet
//
// The control packet is like TCP_ACK, TCP_SYN, TCP_FIN (excluding TCP_RST).
// All these packets do not have payload and the only difference among these is
// the flags.
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);
 }
```

TCP 连接的状态机需要做较多的改动。连接建立过程中，上一个状态发送的包可能会丢失，故超时重传需要在 SYN_SENT 和 SYN_RECV 状态下进行，若收到回应则清空发送队列。此外，纯 ACK 包不需要重传：

```
 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);
       tcp_send_control_packet(tsk, TCP_ACK);
       wake_up(tsk->wait_connect);
    }
    else if (cb->flags == TCP_SYN) {
       tsk->rcv_nxt = cb->seq_end;
       tcp_set_state(tsk, TCP_SYN_RECV);
       tcp_send_control_packet(tsk, TCP_SYN | TCP_ACK);
    }
    else
       log(DEBUG, "Current state is TCP_SYN_SENT but recv not SYN or SYN|ACK");
    break;

 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);

                    tcp_send_control_packet(tsk, TCP_RST);
                    tcp_unhash(tsk);
                    tcp_bind_unhash(tsk);

                    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(tsk->parent->wait_accept);
                }
            }
            else
                log(ERROR, "tsk->parent is NULL\n");
        }
        else
            log(DEBUG, "Current state is TCP_SYN_RECV but recv not ACK");
        break;
```

关闭连接时,需要在 FIN_WAIT_1 和 LAST_ACK 状态下进行超时重传,若收到回应则清空发送队列,关闭定时器,更新状态:

```
    case TCP_FIN_WAIT_1:
        if (!is_tcp_seq_valid(tsk, cb))
```

```
            return;

        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->snd_nxt ==
             tsk->snd_una) {
            tcp_set_state(tsk, TCP_TIME_WAIT);
            tcp_set_timewait_timer(tsk);

            tcp_send_control_packet(tsk, TCP_ACK);

        }
        else if ((cb->flags & TCP_ACK) && tsk->snd_nxt == tsk->snd_una)
            tcp_set_state(tsk, TCP_FIN_WAIT_2);
        else if (cb->flags & TCP_FIN) {
            tcp_set_state(tsk, TCP_CLOSING);
            tcp_send_control_packet(tsk, TCP_ACK);
        }
        break;

    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_CLOSED);
```

```
            tsk->rcv_nxt = cb->seq;
            tsk->snd_una = cb->ack;

            tcp_set_state(tsk, TCP_CLOSED);

            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);

            free_tcp_sock(tsk);
        }
        break;
```

还有 ESTABLISH 状态，先判断收到的序列号是否是预期收到之前的，若是则丢弃。然后判断数据包是否带数据，若带数据则交给处理函数进行处理：先判断数据包长度是否合法，再看缓冲区是否有足够空间，满则等待，否则将数据包加入接收队列，扫描接收队列，若有序列号与 rcv_nxt 匹配的数据包则移动到环形缓冲区，更新接收窗口、发送队列、定时器；若不带数据则验证收到的序列号与期望的是否一致，若收到的更新，则重置定时器，再根据 ACK 更新发送队列和定时器：

```
      case TCP_ESTABLISHED:
         if (less_than_32b(cb->seq, tsk->rcv_nxt)) {
            tcp_send_control_packet(tsk, TCP_ACK);
            return;
         }
         if (!is_tcp_seq_valid(tsk, cb))
            return;
         if (cb->flags & TCP_ACK) {
            tcp_update_window_safe(tsk, cb);
            tsk->snd_una = greater_than_32b(cb->ack, tsk->snd_una) ? cb->ack :
                tsk->snd_una;;
         }
         if (cb->flags & TCP_FIN) {
            tcp_update_send_buffer(tsk, cb->ack);
            tcp_update_retrans_timer(tsk);
            if (tsk->retrans_timer.enable)
               log(ERROR, "still have no ack packet before close wait\n");

            tcp_set_state(tsk, TCP_CLOSE_WAIT);
            handle_tcp_recv_data(tsk, cb);
            tsk->rcv_nxt = cb->seq_end;
            tcp_send_control_packet(tsk, TCP_ACK);
            wake_up(tsk->wait_recv);
         }
         else {
```

```
        if (cb->pl_len != 0)
            handle_tcp_recv_data(tsk, cb);
        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;
            }
            tsk->snd_una = cb->ack;
            tcp_update_window_safe(tsk, cb);
            tcp_update_send_buffer(tsk, cb->ack);
            tcp_update_retrans_timer(tsk);
        }
    }
    break;

// 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);
    tcp_send_control_packet(tsk, TCP_ACK);
    wake_up(tsk->wait_recv);
    pthread_mutex_unlock(&tsk->rcv_buf_lock);
    return 1;
}
```

14.2.2 拥塞控制

1. 发送队列控制与拥塞状态机

由于拥塞窗口的控制由 ACK 包触发，在收到新 ACK 包时，扫描发送队列，将已回复的数据包删除：

```
//Update the TCP send buffer based on the acknowledgment number
int tcp_update_send_buffer(struct tcp_sock *tsk, u32 ack) {
   int ret = 0;
   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)) {
         list_delete_entry(&send_buffer_entry->list);
         free(send_buffer_entry->packet);
         free(send_buffer_entry);
         ret = 1;
      }
   }

   pthread_mutex_unlock(&tsk->send_buf_lock);
   return ret;
}
```

根据不同的拥塞状态，分别处理收到的 ACK 包。在 OPEN 状态下，直接检查拥塞窗口是否小于慢启动阈值，若小于则通过固定步长增加拥塞窗口，否则通过拥塞避免算法增加拥塞窗口。如果是无效 ACK 包，则将状态切换到 DISORDER，且增加 dupacks 计数。

在 DISORDER 状态下，拥塞窗口与上面类似，但是增加了对 dupacks 计数的处理，若 dupacks 计数达到阈值，则判定丢包，启动快重传，且不用等待定时器超时，将状态切换到 RECOVERY，启动快恢复。

在 RECOVERY 状态下，拥塞窗口值变为原来的一半。当对方确认所有进入 RECOVERY 状态前发送的数据包时，将状态切换到 OPEN，否则重传。再若 RECOVERY 状态下收到无效 ACK 包，则 dupacks 计数加一，唤醒发送数据进程。

LOSS 状态仅能通过定时器超时触发，进入时阈值减半，拥塞窗口值变为 1，重新慢启动。当接收方确认所有进入 LOSS 状态前发送的数据包时，将状态切换到 OPEN。若收到无效 ACK 包，则增加 dupacks 计数。

最后更新超时重传定时器。以上过程代码如下：

```
void tcp_congestion_control(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
   int ack_valid = tcp_update_send_buffer(tsk, cb->ack);
```

```
        switch (tsk->c_state) {
            case OPEN:
                if (tsk->cwnd < tsk->ssthresh)
                    tsk->cwnd += 1;
                else
                    tsk->cwnd += 1.0 / tsk->cwnd;
                if (!ack_valid) {
                    tsk->dupacks++;
                    tsk->c_state = DISORDER;
                }
                break;

            case DISORDER:
                if (tsk->cwnd < tsk->ssthresh)
                    tsk->cwnd += 1;
                else
                    tsk->cwnd += 1.0 / tsk->cwnd;
                if (!ack_valid) {
                    tsk->dupacks++;
                    if (tsk->dupacks >= 3) {
                        tsk->ssthresh = max((u32)(tsk->cwnd / 2), 1);
                        tsk->cwnd -= 0.5;
                        tsk->cwnd_flag = 0;
                        tsk->recovery_point = RECOVERY;
                        tcp_retrans_send_buffer(tsk);
                    }
                }
                break;

            case LOSS:
                if (tsk->cwnd < tsk->ssthresh)
                    tsk->cwnd += 1;
                else
                    tsk->cwnd += 1.0 / tsk->cwnd;
                if (ack_valid) {
                    if (cb->ack >= tsk->loss_point) {
                        tsk->c_state = OPEN;
                        tsk->dupacks = 0;
                    }
                }
                else
                    tsk->dupacks++;
```

```
                break;

            case RECOVERY:
                if (tsk->cwnd > tsk->ssthresh && tsk->cwnd_flag == 0)
                    tsk->cwnd -= 0.5;
                else
                    tsk->cwnd_flag = 1;
                if (ack_valid) {
                    if (cb->ack < tsk->recovery_point)
                        tcp_retrans_send_buffer(tsk);
                    else {
                        tsk->c_state = OPEN;
                        tsk->dupacks = 0;
                    }
                }
                else {
                    tsk->dupacks++;
                    wake_up(tsk->wait_send);
                }
                break;

            default:
                break;
        }

        tcp_update_retrans_timer(tsk);
    }
```

2. 拥塞控制实现

这一部分是对之前一些函数的修改。

原有的窗口更新未考虑拥塞窗口，现在需要将拥塞窗口和接收窗口中的最小值作为发送窗口大小：

```
    // update the snd_wnd of tcp_sock
    //
    // if the snd_wnd before updating is zero, notify tcp_sock_send (wait_send)
    static inline void tcp_update_window(struct tcp_sock *tsk, struct tcp_cb *cb)
    {
        u16 old_snd_wnd = tsk->snd_wnd;
        tsk->adv_wnd = cb->rwnd;
        tsk->snd_wnd = min(tsk->adv_wnd, tsk->cwnd*TCP_MSS);
        if (old_snd_wnd == 0)
            wake_up(tsk->wait_send);
    }
```

数据包发送时,需要检测在途数据包和发送窗口大小,若发送窗口不足则等待:

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len) {
   int send_len, packet_len;
   int remain_len = len;
   int handled_len = 0;

   while (!list_empty(&tsk->send_buf))
      sleep_on(tsk->wait_send);

   while (remain_len) {
      send_len = min(remain_len, 1514 - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE -
          TCP_BASE_HDR_SIZE);
      if (tsk->snd_wnd < send_len)
         sleep_on(tsk->wait_send);

      packet_len = send_len + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
          TCP_BASE_HDR_SIZE;
      char *packet = (char *)malloc(packet_len);
      memcpy(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE, buf
          + handled_len, send_len);
      tcp_send_packet(tsk, packet, packet_len);

      int inflight = (tsk->snd_nxt - tsk->snd_una) - tsk->dupacks*TCP_MSS;
      if (max(tsk->snd_wnd-inflight, 0) <= 0) {
         log(DEBUG, "snd_wnd is %d, inflight is %d", tsk->snd_wnd, inflight);
         sleep_on(tsk->wait_send);
      }

      tsk->snd_wnd -= send_len;
      remain_len -= send_len;
      handled_len += send_len;
   }

   return handled_len;
}
```

超时重发也需要考虑拥塞窗口,超时重发时,拥塞状态变为 LOSS,ssthresh 减半,拥塞窗口变为 1,重新慢启动:

```
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 = retranstimer_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_recv);
                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);

                tsk->ssthresh = max((u32)(tsk->cwnd / 2), 1);
                tsk->cwnd = 1;
                tsk->c_state = LOSS;
                tsk->loss_point = tsk->snd_nxt;
                cwnd_record(tsk);

                time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL;
                tcp_retrans_send_buffer(tsk);
            }
        }
    }

    pthread_mutex_unlock(&retrans_timer_list_lock);
}
```

最后，在 TCP 连接的 ESTABLISHED 状态下，使用 `tcp_congestion_control` 函数处理拥塞控制：

```c
        case TCP_ESTABLISHED:
            if (less_than_32b(cb->seq, tsk->rcv_nxt)) {
                tcp_send_control_packet(tsk, TCP_ACK);
                return;
            }

            if (!is_tcp_seq_valid(tsk, cb))
                return;

            if (cb->flags & TCP_ACK) {
                tcp_update_window_safe(tsk, cb);
                tsk->snd_una = greater_than_32b(cb->ack, tsk->snd_una) ? cb->ack :
                    tsk->snd_una;;
            }

            if (cb->flags & TCP_FIN) {
                tcp_update_send_buffer(tsk, cb->ack);
                tcp_update_retrans_timer(tsk);
                if (tsk->retrans_timer.enable)
                    log(ERROR, "still have no ack packet before close wait\n");

                tcp_set_state(tsk, TCP_CLOSE_WAIT);
                handle_tcp_recv_data(tsk, cb);

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

                wake_up(tsk->wait_recv);
            }
            else {
                if (cb->pl_len != 0)
                    handle_tcp_recv_data(tsk, cb);
                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;
                    }
                    tsk->snd_una = cb->ack;
                    tcp_update_window_safe(tsk, cb);
                    tcp_congestion_control(tsk, cb, packet);
                }
            }
```

```
        break;
```

3. 拥塞窗口统计

为了更加准确的记录拥塞窗口的变化，首先在拥塞窗口值变化时记录下来，写入文件。此外，我也创建了一个线程，每隔一段时间读取拥塞窗口值，写入文件：

```
void cwnd_record(struct tcp_sock *tsk) {
    #ifdef LOG_AS_PPT
        struct timeval current;
        gettimeofday(&current, NULL);
        long duration = 1000000 * (current.tv_sec - start.tv_sec) +
            current.tv_usec - start.tv_usec;
        char line[100];
        sprintf(line, "%ld %f %f\n", duration, tsk->cwnd, tsk->cwnd*TCP_MSS);
        fwrite(line, 1, strlen(line), "cwnd.txt");
    #else
        return;
    #endif
    }

    void *tcp_cwnd_thread(void *arg) {
        struct tcp_sock *tsk = (struct tcp_sock *)arg;
        FILE *fp = fopen("cwnd.txt", "w");

        int time_us = 0;
        while (tsk->state == TCP_ESTABLISHED && time_us < 1000000) {
            usleep(500);
            time_us += 500;
            fprintf(fp, "%d %f %f\n", time_us, tsk->cwnd, tsk->cwnd*TCP_MSS);
        }
        fclose(fp);
        return NULL;
    }
```

14.3 实验结果

14.3.1 丢包恢复

1. 本实验 server 与本实验 client 进行数据传输，用 md5sum 比较两个文件是否完全相同，结果如下：

图 **14.1.** my_server

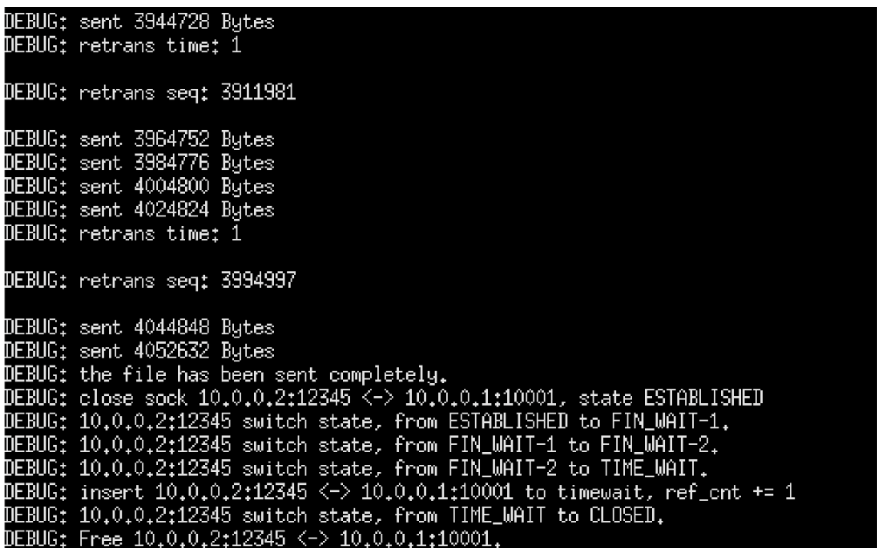


图 **14.2.** my_client

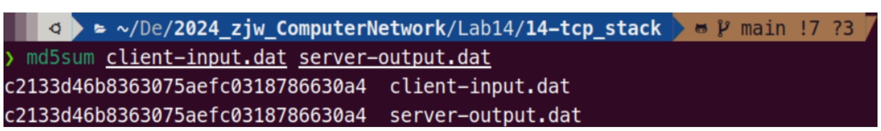


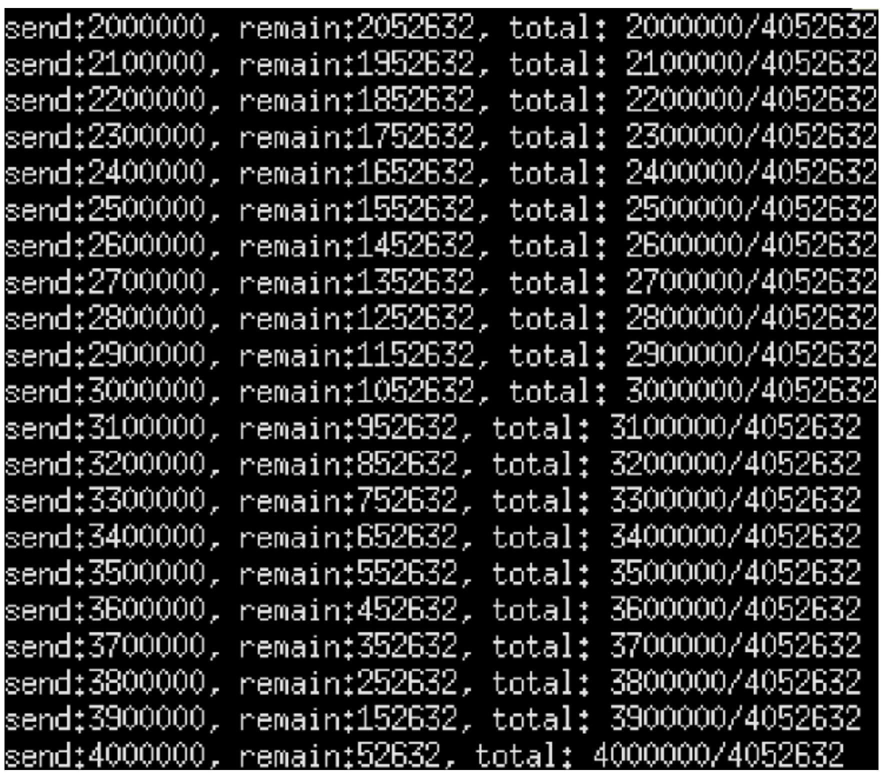
图 **14.3.** md5sum

可以看出,出现了丢包,但是通过重传机制,最终文件完全相同。

2. 本实验 server 与标准 client 进行数据传输,用 md5sum 比较两个文件是否完全相同,结果如下:

```
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.
```

图 **14.4.** my_server



```
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
```

图 **14.5.** std_client

```
~/De/2024_zjw_ComputerNetwork/Lab14/14-tcp_stack  main !7 ?3
❯ md5sum client-input.dat server-output.dat
4bb8a034cd44c20105bd3252d9c6495f  client-input.dat
4bb8a034cd44c20105bd3252d9c6495f  server-output.dat
```

图 **14.6.** md5sum

可以看出，出现了丢包，但是通过重传机制，最终文件完全相同。

3. 标准 server 与本实验 client 进行数据传输，用 md5sum 比较两个文件是否完全相同，结果如下：

```
('10.0.0.2', 12345)
```

图 **14.7.** std_server

```
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.
```

图 **14.8.** my_client

```
~/De/2024_zjw_ComputerNetwork/Lab14/14-tcp_stack  main !7 ?3
❯ md5sum client-input.dat server-output.dat
f1a72ea92a9d8735d3a20debdface99e  client-input.dat
f1a72ea92a9d8735d3a20debdface99e  server-output.dat
```

图 **14.9.** md5sum

可以看出，出现了丢包，但是通过重传机制，最终文件完全相同。

14.3.2 拥塞控制

拥塞窗口随时间的变化图如下：

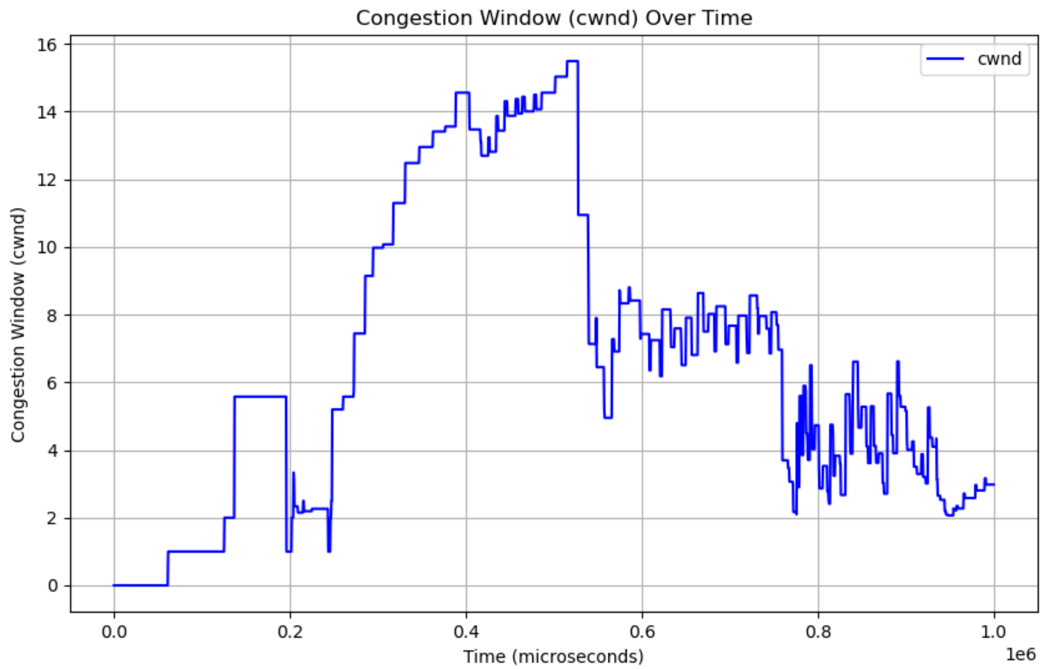


图 **14.10.** 拥塞窗口随时间的变化

可以看出,拥塞窗口的变化符合慢启动、拥塞避免、快恢复、快重传的规律。若拥塞窗口下降再迅速恢复,说明这是一次快重传快恢复的过程。若拥塞窗口下降到最低点再缓慢增长,说明这是一次超时重传的过程。

14.4 实验总结

本次实验中,我们实现了丢包恢复和拥塞控制的功能。通过实验,我们再次深入了解了 TCP 协议栈的工作原理。实现可靠传输使得数据传输更加稳定,实现拥塞控制使得网络更加稳定。通过本次实验,我们对 TCP 协议栈的实现有了更深入的了解,更加接近真实的 TCP 协议栈。