网络实验报告

雷正宇 2016K8009909005

上周实验补充部分

上周实验我没有完成需求，在传输文件大于 50KB 时会出错。我找到了原因：tcp_sock_write 函数在写入时参考的 snd_wnd 不一定是对方真正的接收窗口，而只是发送方推测出的窗口。错误发生的情景是：发送方不断调用 tcp_sock_write 函数写入数据并发送，直到将 snd_wnd 用完进入等待队列；接收方在经过一段时间后才接收到第一个包，然后进行读取并回复 ACK，此时唤醒发送方。这时，发送方以为接收窗口还很大，于是继续不断发送，而事实上，接收窗口要比发送方以为的要小得多，因为接收方还只针对发送方最初的包进行了回复，tcp_sock_read 函数执行线程没有跟上写 buffer 的速度；这时，发送方以为自己发送的包是安全的（接收窗口够大），而事实上接收窗口很小，一旦 read 线程没有跟上 write 的速度，write 函数就会把 buffer 写满，造成错误。

解决的方式是，我在 tcp_sock 结构中加入了一个写 buffer 等待队列：

```
1    struct synch_wait *wait_write;
```

当套接字接收到新的包后，判断 buffer 是否有足够的空间，如果有，就写入，否则就等待：

```
1  if (cb->pl_len > 0) {
2    wake_up(tsk->wait_recv);
3    while (less_than_32b(ring_buffer_free(tsk->rcv_buf), cb->pl_len)) {
4      sleep_on(tsk->wait_write);
5    }
6    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
7  }
```

tcp_sock_read 函数在进行读取后唤醒这个写等待队列：

```
1  int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)
2  {
3    while (ring_buffer_empty(tsk->rcv_buf)) {
4      if (tsk->state != TCP_ESTABLISHED) {
5        return 0;
6      }
7      sleep_on(tsk->wait_recv);
```

```
    }
    int read_len = read_ring_buffer(tsk->rcv_buf, buf, len);
    wake_up(tsk->wait_write);
    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);

    return read_len;
}
```

这样就能正常处理大包了。

实验内容

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

实验流程

当有 SYN 或者 FIN 包丢失时，接收方没有回应 ACK，发送方根据超时重传机制重新发送 SYN 或 FIN 包。

在这个过程中，接收方不用额外处理事情。发送方在第一次发送 SYN 或 FIN 包之后状态已经改变，重发工作由 send_buf 独自完成。以下是 send_buf 的具体实现：

```
enum buf_kind { DATA, SYN, FIN };
struct send_buf_packet {
```

```
3       int len;
4       char *packet; // the whole tcp packet
5       struct list_head list;
6       u32 seq_end;
7       enum buf_kind kind;
8   };
9
10  void send_buf_add(struct tcp_sock *tsk, char *packet, int len, enum
    buf_kind kind, u32 seq_end)
11  {
12      struct send_buf_packet *new_packet = malloc(sizeof(struct
    send_buf_packet));
13      new_packet->packet = malloc(len);
14      new_packet->len = len;
15      new_packet->kind = kind;
16      new_packet->seq_end = seq_end;
17      memcpy(new_packet->packet, packet, len);
18      list_add_tail(&new_packet->list, &tsk->send_buf);
19      if (tsk->retrans_timer.enable == 0) {
20          tcp_set_retrans_timer(tsk);
21      }
22  }
23
24  void send_buf_remove(struct send_buf_packet *sbp)
25  {
26      list_delete_entry(&sbp->list);
27      free(sbp->packet);
28      free(sbp);
29  }
30
31  void send_buf_retrans(struct send_buf_packet *sbp)
32  {
33      log(DEBUG, "retrans: %d", sbp->seq_end);
34      char *packet = malloc(sbp->len);
35      memcpy(packet, sbp->packet, sbp->len);
36      ip_send_packet(packet, sbp->len);
37  }
38
39  void send_buf_ack(struct tcp_sock *tsk, struct tcp_cb *cb)
40  {
41      struct send_buf_packet *sbp, *q;
42      list_for_each_entry_safe(sbp, q, &tsk->send_buf, list) {
43          if (less_or_equal_32b(sbp->seq_end, cb->ack)) {
44              send_buf_remove(sbp);
45              update_retrans_timer(tsk);
```

```
46          }
47      }
48  }
```

每次发送 SYN 和 FIN 包时，都调用 send_buf_add 函数将要发送的包的副本保存在 send_buf 中。如果定时器没有启动，就启动定时器。每次收到 ACK 包就调用 send_buf_ack 函数将 send_buf 中待确认的包移除。当定时器到时事件发生，就调用 send_buf_retrans 函数重传一次副本。这里没有记录重传的次数，如果要记也很简单，在 send_buf_packet 结构体中添加一个 cnt 字段就可以了。

以下是发送 SYN 和 FIN 包时的处理：

```
1  void tcp_send_control_packet(struct tcp_sock *tsk, u8 flags)
2  {
3    ...
4    if (flags & (TCP_SYN|TCP_FIN)) {
5      tsk->snd_nxt += 1;
6      if (flags & TCP_SYN) {
7        send_buf_add(tsk, packet, pkt_size, SYN, tsk->snd_nxt);
8      } else {
9        send_buf_add(tsk, packet, pkt_size, FIN, tsk->snd_nxt);
10     }
11   }
12
13   ip_send_packet(packet, pkt_size);
14 }
```

接收队列的处理上如果不用到优先队列或者其它堆排序算法的话，单纯用链表实现效率会很低。本次实验时间实在紧张，只能用链表实现了一个，原理如下：

每次收到新的 data 包，判断是否乱序，如果是乱序包，就用 recv_ofo_buf_add 函数将它的副本放入 rcv_ofo_buf 队列：

```
void recv_ofo_buf_add(struct tcp_sock *tsk, char *packet, int len, u32
seq, u32 seq_end)
{
    struct recv_ofo_buf_packet *robp = malloc(sizeof(struct
recv_ofo_buf_packet));
    robp->len = len;
    robp->seq = seq;
    robp->seq_end = seq_end;
    robp->packet = malloc(len);
    memcpy(robp->packet, packet, len);
    list_add_tail(&robp->list, &tsk->rcv_ofo_buf);
}
```

乱序队列的实现中，保存了每个乱序包的首尾序号（尾序号是最后一个字节的下一个字节号）。这样做是为了方便检查哪些包是连在一起的。

如果不是乱序的，就直接将包写入 ring_buffer 供用户读取，还没完！由于现在有了乱序队列，所以每次写入 ring_buffer 后都要对乱序队列进行一次检查，检查的参数就是写入 ring_buffer 时的末尾序号，当有乱序包的头序号和已经写入的尾序号相等时就对该包进行再读取：

```
void recv_ofo_buf_check(struct tcp_sock *tsk, u32 seq)
{
    struct recv_ofo_buf_packet *robp, *q;
    list_for_each_entry_safe(robp, q, &tsk->rcv_ofo_buf, list) {
        if (robp->seq == seq) {
            recv_ofo_buf_reload(tsk, robp);
            recv_ofo_buf_check(tsk, robp->seq_end);
            return;
        } else if (less_than_32b(robp->seq, seq)) {
            recv_ofo_buf_remove(robp);
        }
    }
}

void recv_ofo_buf_reload(struct tcp_sock *tsk, struct
recv_ofo_buf_packet *robp)
{
    insert_to_ring(tsk, robp->packet, robp->len);
    recv_ofo_buf_remove(robp);
}

void recv_ofo_buf_remove(struct recv_ofo_buf_packet *robp)
{
```

```
23      list_delete_entry(&robp->list);
24      free(robp->packet);
25      free(robp);
26  }
```

这个 check 过程用了递归的写法：当成功将乱序队列里的一个乱序包的内容读取到 ring_buffer 后，根据这个乱序包的末尾序号再进行递归检查，从而把所有恢复顺序的包都进行再读取。

遗留问题

以上实现有一个十分严重的问题，直接导致实验结果低成功率！接收方处理数据包的过程是这样的：

```
1  tcp_update_window_safe(tsk, cb);
2  if (out_of_order) {
3    recv_ofo_buf_add(tsk, packet, cb->pl_len, cb->seq, cb->seq_end);
4    tcp_send_control_packet(tsk, TCP_ACK);
5  } else {
6    insert_to_ring(tsk, cb->payload, cb->pl_len);
7    tcp_send_control_packet(tsk, TCP_ACK);
8    recv_ofo_buf_check(tsk, cb->seq_end);
9  }
```

要想在判断数据包乱序后将现场信息保存并塞进乱序队列是很难的事情，这就意味着每次回复 ACK 的包中控制信息都不是针对特定乱序包的，而是此刻的。因此为了方便，我将接收方逻辑做成了选择确认：每次收到包后都回复对应 ACK，即便是乱序包，也回复 ACK。但是我的发送方仍然是累积确认：发送方在收到 ACK 后利用这个布尔表达式来移除数据包：

```
1  less_or_equal_32b(sbp->seq_end, cb->ack)
```

时间实在紧张，目前（交作业时）没有针对这个问题进行修正。

实验结果及分析

修改后的 tcp_stack 依然能正常完成数据传输的任务和简单的重传，但是在高丢包率（我设置为 10%）时，总会发生错误。