

# 网络机制实验二

范子墨

2021K8009929006

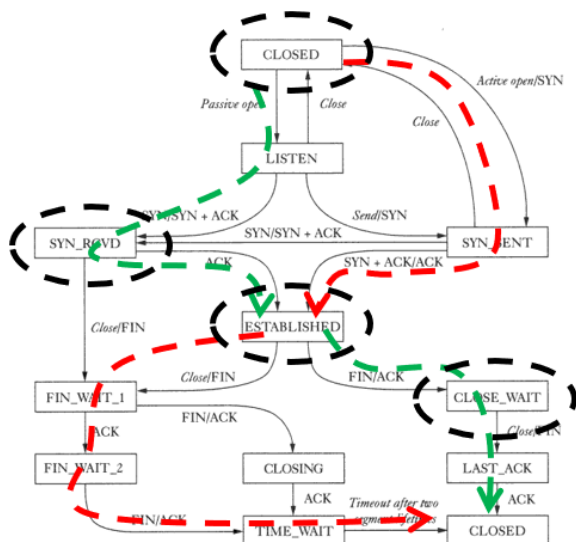
## 一、实验内容

- 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

## 二、设计思路

### 1、总体设计思路

在上一次的实验基础上实现可靠传输, 节点之间在有丢包网络中能够建立连接并正确传输数据。当发送数据包时, 要将数据包放入 `send_buf` 缓冲区, 即对于所有需要发送数据包时候, 将数据包放入 `send_buf` 中, 即调用 `new_data_block` 函数后将该结构加入到队列中, 并设置定时器。在收到 ACK 后, 调用 `tcp_free_send_buf` 函数检查 `ack` 大小, 删除 `send_buf` 队列中已经确认过的数据包, 并删除定时器。当接收数据包时, 检查是否符合当前期望序列号, 若符合, 则唤醒 `wait_recv` 对应线程, 并写入接收缓冲区, 否则加入 `rcv_ofo_buf` 缓冲区中。



### 2、send\_buf 队列维护

首先是给 `send_buf` 和 `rcv_ofo_buf` 重新定义一个新的数据结构, 将所有数据包形成链表。数据结构中需要包含 `flags`, 发送计数, 长度和结束序列号以及数据包内容。

```
struct data_packet {
```

```

struct list_head list;
u8  flags;
u8  times;
u32 seq;
u32 len;
u32 seq_end;
char *packet;
};

```

而后对于所有需要发送数据包的地方进行修改，将数据包加入发送队列。即调用 `new_data_block` 函数。该函数有四个参数，`flags` 主要是用于判断是否为 SYN 或者 FIN 包，`seq` 是序列号，`len` 和 `buf` 为数据包的传输数据长度和内容。

```

struct data_packet *new_data_block(u8 flags, u32 seq, u32 len, char *buf)

```

在该函数中，会构建一个 `data_packet` 结构体，将该结构体中的变量赋值后返回。这里需要注意结束序列号中要考虑到 SYN 或者 FIN 标志。如果是 SYN 或者 FIN 的话，由于他们虽然不携带数据（但是内核版本的 FIN 包是携带数据的），但是 TCP 数据流的一部分，因此占用一个序列号，结束序列号要加一。

```

dp->seq_end = seq + len + ((flags & (TCP_SYN|TCP_FIN)) ? 1 : 0);

```

在 `tcp_sock_connect` 函数中发送 SYN，因此需要将该包加入发送队列中，不包含数据。

```

struct data_packet *dp = new_data_block(TCP_SYN, tsk->snd_nxt, 0, NULL);
pthread_mutex_lock(&send_buf_lock);
list_add_tail(&dp->list, &tsk->send_buf.list);
pthread_mutex_unlock(&send_buf_lock);

```

在 `tcp_sock_close` 中同样发送 FIN|ACK 包，将其加入发送队列中，这里不再赘述。

在 `tcp_sock_write` 函数中，由于非 SYN 或者 FIN 包，不妨将 `flags` 设为 ACK，其中注意 `pt` 为已经发送数据的长度，因此 `buf+pt` 即将指针偏移至当前要发送的数据包的起始点。

```

struct data_packet *dp = new_data_block(TCP_ACK, seq, data_len, buf + pt);

```

其余发送数据包的地方同理。

当收到 ACK 后，需要将发送队列中对应的数据包删除。即在 `tcp_process` 函数中对应收到 ACK 包后，调用 `tcp_free_send_buf` 函数。在该函数中，遍历发送队列，比较当前数据包的序列号和发送队列中的结束序列号，如果结束序列号在当前数据包的序列号之前，说明该数据块已经收到 ACK 成功被接收了，则删除该条目，同时取消重传定时器。由于部分数据已经被确认，则重启定时器，设置时间为 200ms。

```

list_for_each_entry_safe(tmp, q, &tsk->send_buf.list, list)
{
    if(tmp->seq_end <= cb->ack)
    {
        list_delete_entry(&tmp->list);
        free(tmp->packet);
        free(tmp);
        tcp_unset_retrans_timer(tsk);
        if(!list_empty(&tsk->send_buf.list))
            tcp_set_retrans_timer(tsk);
    }
}

```

### 3、rcv\_ofo\_buf 队列维护

根据 wireshark 抓包来看，client 端发送数据包为 PSH|ACK 包，因此可以将对于数据包的处理同样放入对 ACK 包的处理中。如果期望接收到的序列号与当前收到的序列号，则调用 tcp\_rcv\_ofo\_pkt 函数尽心处理，并回复 ACK。

```
if(tsk->rcv_nxt != cb->seq)
{
    tcp_rcv_ofo_pkt(tsk, cb);
    tcp_send_control_packet(tsk, TCP_ACK);
    break;
}
```

在 tcp\_rcv\_ofo\_pkt 函数中，同样调用 new\_data\_block 函数构建一个 data\_packet 结构体，并将该条目插入序列号对应的位置，即乱序接收队列是按照序列号从小到大排列的。

```
list_for_each_entry_safe(tmp, q, &tsk->rcv_ofo_buf.list, list)
{
    if(tmp->seq_end > cb->seq_end)
        break;
}
```

而收到数据包后，如果满足接收到的序列号与期望收到的序列号相等，则遍历乱序接收队列，如果有符合期望接收到的下一个序列号等于条目对应的序列号的（此时期望接收到的序列号已经更新），则将该条目的数据包写入环形缓冲区，并删除该序列号。这里处理的一个基本逻辑是，每个数据包的长度是固定的，即发送内容如何分割是固定的，因此可以直接有序列号与期望收到的序列号是否相等判断，而不需要进行大小判断。此外，定义 fin\_flag，即如果同时是 FIN 包，则发送 ACK 回复。（内核版本的代码会出现 FIN 包中包含数据的情况）

```
list_for_each_entry_safe(tmp, q, &tsk->rcv_ofo_buf.list, list)
{
    if(tmp->seq == tsk->rcv_nxt)
    {
        wake_up(tsk->wait_rcv);
        pthread_mutex_lock(&rcv_buf_lock);
        write_ring_buffer(tsk->rcv_buf, tmp->packet, tmp->len);
        tsk->rcv_wnd -= tmp->len;
        pthread_mutex_unlock(&rcv_buf_lock);
        tsk->rcv_nxt = tmp->seq_end;
        fin_flag = tmp->flags & TCP_FIN;
        list_delete_entry(&tmp->list);
        free(tmp->packet);
        free(tmp);
    }
    else
        break;
}
```

#### 4、定时器维护

与维护发送队列类似，所有发送数据包的地方同样需要设置定时器，即如果没有启用重传定时器，则调用 `tcp_set_retrans_timer` 函数启动重传定时器。

```
if(!tsk->retrans_timer.enable)
```

```
    tcp_set_retrans_timer(tsk);
```

`tcp_set_retrans_timer` 函数中，设置类型和 `enable` 为 1，超时时间为 `TCP_RETRANS_INTERVAL_INITIAL`，并将其加入到定时器队列中，注意这里的定时器队列与 `TIME_WAIT` 定时器队列相同，只需在 `tcp_scan_timer_list` 中加以区别后处理即可。

`tcp_scan_timer_list` 函数中，遍历定时器列表，在原来的基础上添加判断 `type=1` 的情况，如果 `type=1`，如果重传次数已经超过三次，则发送 RST 包。其余情况按照原来发送数据包的方式即可。

```
if(dp->flags & (TCP_SYN | TCP_FIN))
```

```
    {
        tcp_send_control_packet(tsk, dp->flags);
    }
```

```
    else
```

```
    {
```

```
        int pkt_len = dp->len + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
        TCP_BASE_HDR_SIZE;
```

```
        char *packet = malloc(pkt_len);
```

```
        memcpy(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
        TCP_BASE_HDR_SIZE, dp->packet, dp->len);
```

```
        tcp_send_packet(tsk, packet, pkt_len);
```

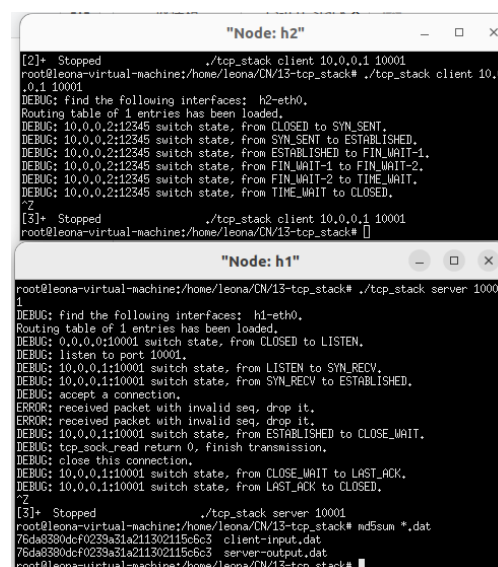
```
    }
```

同时由每次重传定时器时间翻倍，更新超时时间。

```
p->timeout = TCP_RETRANS_INTERVAL_INITIAL * (1 << (dp->times + 1));
```

### 三、实验结果

#### 1、自己对自己传输



```
"Node: h2"
[2]+ Stopped ./tcp_stack client 10.0.0.1 10001
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces: h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to 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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
^Z
[3]+ Stopped ./tcp_stack client 10.0.0.1 10001
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack#

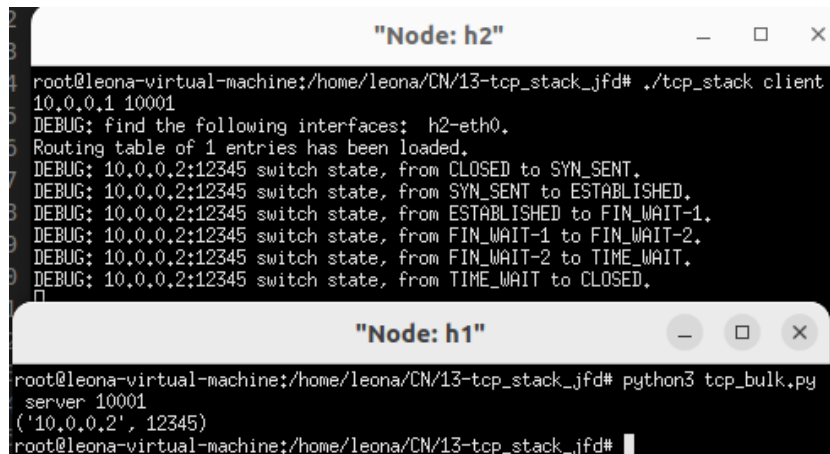
"Node: h1"
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from LISTEN to SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
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.
^Z
[3]+ Stopped ./tcp_stack server 10001
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack# md5sum *.dat
76da8380dcf0239a31a2113021156c3 client-input.dat
76da8380dcf0239a31a2113021156c3 server-output.dat
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack#
```

Figure 1：自己对自己

```
leona@leona-virtual-machine:~/CN/14_tcp_stack_fzm$ md5sum *.dat
76da8380dcf0239a31a211302115c6c3  client-input.dat
76da8380dcf0239a31a211302115c6c3  server-output.dat
```

Figure 2: 传输结果验证

2、一段为内核版本一端为自己的代码



```

"Node: h2"
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack_jfd# ./tcp_stack client
10.0.0.1 10001
DEBUG: find the following interfaces: h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to 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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.

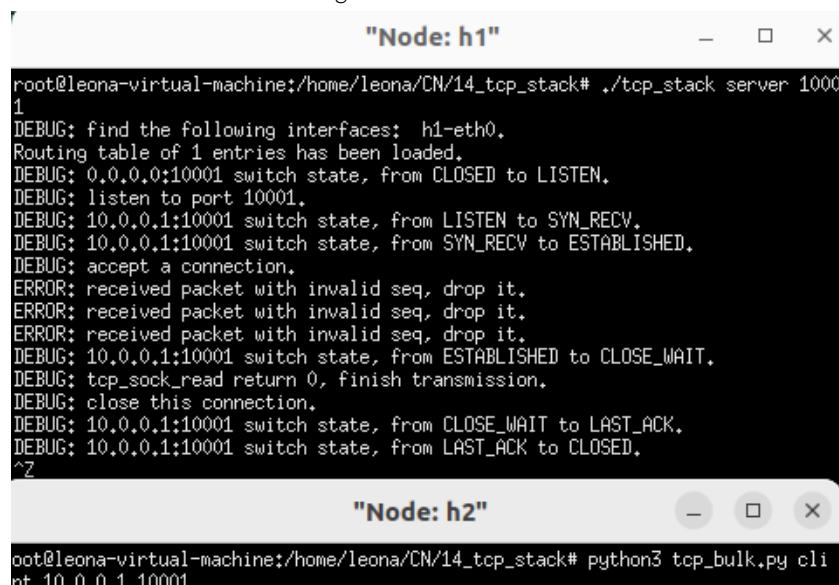
"Node: h1"
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack_jfd# python3 tcp_bulk.py
server 10001
('10.0.0.2', 12345)
root@leona-virtual-machine:/home/leona/CN/13-tcp_stack_jfd#

```

Figure 3: server 端为 py 脚本

```
leona@leona-virtual-machine:~/CN/14_tcp_stack_fzm$ md5sum *.dat
76da8380dcf0239a31a211302115c6c3  client-input.dat
76da8380dcf0239a31a211302115c6c3  server-output.dat
```

Figure 4: 结果验证



```

"Node: h1"
root@leona-virtual-machine:/home/leona/CN/14_tcp_stack# ./tcp_stack server 1000
1
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 10.0.0.1:10001 switch state, from LISTEN to SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
ERROR: received packet with invalid seq, drop it.
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.
^Z

"Node: h2"
root@leona-virtual-machine:/home/leona/CN/14_tcp_stack# python3 tcp_bulk.py cli
ent 10.0.0.1 10001

```

Figure 5: client 端为 py 脚本

```
leona@leona-virtual-machine:~/CN/14_tcp_stack_fzm$ md5sum *.dat
76da8380dcf0239a31a211302115c6c3  client-input.dat
76da8380dcf0239a31a211302115c6c3  server-output.dat
```

Figure 6: 结果验证

#### 四、实验总结

本次代码量不大，主要是对于如何妥善两个队列以及计时器，在设计的时候比较耗费脑筋。本次实验让我对超时重传和 TCP 协议有了更深入的理解。