计算机网络实验 13 报告

学号　2017K8009929059
姓名　於修远

网络传输机制实验三

一、实验内容

- 完成接收队列和发送队列的相关内容。
- 实现超时定时器及重传机制。
- 为 TCP 传输增加有丢包情况下的控制转换和数据处理。

二、实验流程

（一）收发队列实现

1、接收队列处理

接收队列的主体在上次实验中已经实现。由于重传等的存在，本次实验只需引入 ofo 队列即可。Ofo 队列用于保存接收到的所有 seq 不连续的 PSH 包，队列中的元素按 seq 大小进行排列。如下图所示为对于 PSH|ACK 包进行处理的新逻辑。第一步是检查新到达包的 seq 是否为连续的下一个待接收包。由于重传包应当忽略，所以不考虑新 seq 小于下一待接收序号的情况。若二者相等，意味着这个包可以直接写入，并更新下一待接收序列号。与此同时，原先缓存在 ofo 队列中的数据包可能成为了满足写入条件的连续包，所以需要对其进行一次扫描，将其中所有连续的内容加入接收队列，并更新待接收序号。最后，如果新到达包的 seq 大于下一待接收序号，则将其写入 ofo 缓存队列。

```
u32 seq_end = tsk->rcv_nxt;
if (seq_end == cb->seq) {
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    seq_end = cb->seq_end;
    struct ofo_buffer *ofo, *q;
    list_for_each_entry_safe(ofo, q, head: &tsk->rcv_ofo_buf, list) {
        if (seq_end < ofo->seq) break;
        else {
            seq_end = ofo->seq_end;
            write_ring_buffer(ofo->tsk->rcv_buf, ofo->payload, ofo->pl_len);
            list_delete_entry(&ofo->list);
            free(ofo->payload);
            free(ofo);
        }
    }
    tsk->rcv_nxt = seq_end;
} else if (seq_end < cb->seq) {
    write_ofo_buffer(tsk, cb);
}
```

由于 cb 模块会在每次处理后被释放，所有 ofo 队列中需要缓存其中的关键信息，包括 seq 和 seq_end，以及数据内容和其长度，此外还需要记录与之关联的 socket 信息。添加 ofo 队列记录的核心代码如下图所示。简而言之，在新插入一个 ofo 记录时，需要遍历原有的 ofo 缓存队列，将其按序插入；若遍历队列后还未插入，则插入队尾。需要注意的是可能存在不连续包重复到达的情况。所以如果探测到写入包与队列中原有的包发生了重复，应当直接丢弃处理中的包，结束写入。

```
int insert = 0;
struct ofo_buffer *pos, *last = &head_ext;
list_for_each_entry(pos, head: &tsk->rcv_ofo_buf, list) {
    if (cb->seq > pos->seq) {
        last = pos;
        continue;
    } else if (cb->seq == pos->seq) return;
    list_insert(&buf->list, &last->list, &pos->list);
    insert = 1;
    break;
}
if (!insert) list_add_tail(&buf->list, &tsk->rcv_ofo_buf);
```

2、发送队列处理

每个发出的数据包，只要占有独立的 seq 信息（SYN 包或 FIN 包或非控制包），就需要被加入发送记录。具体而言，该过程包括了两个方面：加入发送包队列，该队列主要保存了发送包的必要信息及 seq 等判定标志；队列长度增加 1，主要用于控制在途数据量，也可以改用首 seq 到末 seq_end 的差值来衡量。

```
static void put_sendbuf(struct tcp_sock *tsk, char *packet, int size) {
    char *temp = (char *) malloc( size: sizeof(char) * size);
    memcpy(temp, packet, n: sizeof(char) * size);
    struct send_buffer *buf = (struct send_buffer *) malloc(sizeof(struct send_buffer));
    buf->packet = temp;
    buf->len = size;
    buf->seq_end = tsk->snd_nxt;
    buf->times = 1;
    buf->timeout = TCP_RETRANS_INTERVAL_INITIAL;
    pthread_mutex_lock(&tsk->send_lock);
    list_add_tail(&buf->list, &tsk->send_buf);
    pthread_mutex_unlock(&tsk->send_lock);
    pthread_mutex_lock(&tsk->count_lock);
    tsk->send_buf_count++;
    pthread_mutex_unlock(&tsk->count_lock);
}
```

新数据包到达后，首先检查其中的 ack 字段是否有效。若是，则遍历发送队列，将其中已被确认的包（即 seq_end 字段不超过 ack 值的包）全部移除，并更新已确认序列号。

```
if ((cb->flags) | TCP_ACK) {
    struct send_buffer *buf, *q;
    list_for_each_entry_safe(buf, q, head: &tsk->send_buf, list) {
        if (buf->seq_end > cb->ack) break;
        else {
            tsk->snd_una = buf->seq_end;
            tcp_pop_sendbuf(tsk, buf);
        }
    }
}
```

发送记录的移除逻辑与插入相仿，包括移出发送序列并释放空间和队列长度自减 1。需要注意的是二者都需要在互斥锁内完成。

```
void tcp_pop_sendbuf(struct tcp_sock *tsk, struct send_buffer *buf) {
    pthread_mutex_lock(&tsk->send_lock);
    list_delete_entry(&buf->list);
    free(buf->packet);
    free(buf);
    pthread_mutex_unlock(&tsk->send_lock);
    pthread_mutex_lock(&tsk->count_lock);
    tsk->send_buf_count--;
    pthread_mutex_unlock(&tsk->count_lock);
}
```

（二）定时器设计

定时器设计的核心是定时扫描函数。在本次实验中，扫描器的唤醒间隔改为 10ms 一次，并加入了对 type 为 1 的记录的支持，这些记录即为重传记录。重传的起始等待时间为 200ms，每重发一次等待时间就会翻倍。扫描从每个 socket 被创建就开始进行，直到 socket 被释放时结束。

Type 为 1 的定时器记录本质上是一个到 socket 的映射。线程中遍历地检查该 socket 绑定的发送队列，发送队列中的每条记录都有一个独立的超时计时器，每当等待时间归 0 就重发一次数据包。具体实现如下图。

```
list_for_each_entry(buf, head: &tsk->send_buf, list) {
    buf->timeout -= TCP_TIMER_SCAN_INTERVAL;
    if (!buf->timeout) {
        if (buf->times++ == 3) {
            printf( format: "Packet Loss!!!\n");
            buf->times = 1;
            buf->timeout = TCP_RETRANS_INTERVAL_INITIAL;
        } else {
            char *temp = (char*)malloc( size: buf->len * sizeof(char));
            memcpy(temp, buf->packet, buf->len);
            ip_send_packet(temp, buf->len);
            if (buf->times == 2) buf->timeout = 2 * TCP_RETRANS_INTERVAL_INITIAL;
            else buf->timeout = 4 * TCP_RETRANS_INTERVAL_INITIAL;
        }
    }
}
```

（三）丢包处理分析

1、控制包丢包的情况

首先考虑连接建立阶段的丢包。若是发起方的 SYN 丢包，由于无回复，发起方会重传 SYN，合理。若是回复的 SYN|ACK 丢失，由于无回复，发起方会重传 SYN，但由于 seq 小于新的 seq_end，所以不会引起错误；被动方未收到回复，所以会重传 SYN|ACK，合理。若是发起方回复的 ACK 丢失，被动方无法跟进状态，但可以通过后续包来识别，例如收到 PSH 包时强制切换为 TCP_ESTABLISHED 状态，如下图。

```
case (TCP_PSH | TCP_ACK):
    if (tsk->state == TCP_SYN_RECV) tcp_set_state(tsk, state: TCP_ESTABLISHED);
```

再考虑断开连接时的丢包。第一次握手 FIN 包丢失，与 SYN 包丢失类似。第二三次握手的 ACK 被丢弃时可以靠 FIN 补救，直接强制切换；FIN 由于会进入发送队列，所以也不用担心丢包的情况。唯一存在问题的是最后的 ACK 丢失，此时断开发起端进入 TIME_WAIT 状态，对端却仍然在 LAST_ACK 状态。解决方法是等待时间后直接自行关闭连接，如下图所示。

```
case TCP_FIN:
    switch (tsk->state) {
        case TCP_ESTABLISHED:
            tcp_set_state(tsk, state: TCP_LAST_ACK);
            tcp_send_control_packet(tsk, flags: TCP_ACK | TCP_FIN);
            tcp_set_timewait_timer(tsk);
            break;
```

2、数据包丢包的情况

数据包丢包的情况较为简单。如下图所示，在` tcp_sock_write `函数中将强制挂起改为检查发送队列长度超过 5 时挂起，从而实现了在途数据包的上限控制；如果某数据包发生了丢包，或是其对应的回复 ack 发生了丢包，都会导致发送被阻塞。此时，被阻塞的包会在重传队列中被反复重传，直至收到正确的 ack 回复。

```
pthread_mutex_lock(&tsk->count_lock);
while (tsk->send_buf_count >= 5) {
    pthread_mutex_unlock(&tsk->count_lock);
    sleep_on(tsk->wait_send);
    pthread_mutex_lock(&tsk->count_lock);
}
pthread_mutex_unlock(&tsk->count_lock);
```

三、结果分析

如下图，执行` tcp_topo_loss `脚本后，将 h1 节点作为 server 端，将 h2 节点作为 client 端，建立 tcp 连接。两节点间的交互结果如下图所示，可见传输过程正常。` md5sum `的比对结果也显示接收到的文件无误。换用上次实验的` tcp_stack_file.py `脚本作为服务器端或客户端，结果也都相同。

```
"Node: h1"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj13# ./tcp_stack server 100
01
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 CLOSED to SYN_RECV.
ckpt!
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
Begin to receive server-output.dat from client.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
"Node: h2"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj13# ./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.
Begin to transfer client-input.dat to server.
Transfer finished.
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: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
^[[A^C
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj13# md5sum client-input.da
t
e027a700c6447240ea56a25facb8f8bf  client-input.dat
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj13# md5sum server-output.d
at
e027a700c6447240ea56a25facb8f8bf  server-output.dat
```