# 计算机网络实验 12报告

2019K8009907018 王畅路

网络机制传输实验

## 实验内容

- 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。
- 实现拥塞控制功能。
- 在给定拓扑下验证拥塞控制的正确性。

## 实验过程

1.引入send_buffer和ofo_buffer

每一次发包后将发送包对应消息的内容以及相关flag等信息存入发送队列中。在接受到新包后，抛弃掉seq小于当前下一个要接受序号的包，并将相等的包写入ring_buffer，大于的包写入ofo队列。如果有新的包写入ringbuffer，需要扫描接受队列并将其中与刚刚写入的包序号连续的包写入ringbuffer并更新rcv_nxt信息。

```
typedef struct send_buffer_entry {
    struct list_head list;
    char *packet;
    int len;
} send_buffer_entry_t;

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

需要注意的是这里写入ofo队列需要按序写入，实现代码如下：

```
void tcp_add_recv_ofo_buffer(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    recv_ofo_buf_entry_t *recv_entry = (recv_ofo_buf_entry_t
*)malloc(sizeof(recv_ofo_buf_entry_t));
    recv_entry->seq = cb->seq;
    recv_entry->len = cb->pl_len;
    recv_entry->data = (char *)malloc(cb->pl_len);
    memcpy(recv_entry->data, cb->payload, cb->pl_len);
    init_list_head(&recv_entry->list);

    recv_ofo_buf_entry_t *entry, *entry_q;
    list_for_each_entry_safe (entry, entry_q, &tsk->rcv_ofo_buf, list) {
        if (recv_entry->seq == entry->seq) {
            return;
        }
        if (less_than_32b(recv_entry->seq, entry->seq)) {
```

```
            list_add_tail(&recv_entry->list, &entry->list);
            return;
        }
    }
    list_add_tail(&recv_entry->list, &tsk->rcv_ofo_buf);
}
```

2.添加状态处理逻辑

为了实现对不符合seq的包进行丢弃的逻辑，对之前的状态机处理逻辑进行重构，由根据当前的状态来对传入的包进行判断改为由传入的包的类型再根据当前状态来进行状态转移的判断。在established状态中，将符合条件seq的包加入ringbuffer中，并缓存将要接受的包。

```
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 *entry, *q;
    list_for_each_entry_safe(entry, q, &tsk->rcv_ofo_buf, list)
    {
        if (seq_end < entry->seq)
            break;
        else
        {
            seq_end = entry->seq_end;
            write_ring_buffer(entry->tsk->rcv_buf, entry->payload, entry - > pl_len);
            list_delete_entry(&entry->list);
            free(entry->payload);
            free(entry);
        }
    }
    tsk->rcv_nxt = seq_end;
}
else if (seq_end < cb->seq)
    write_ofo_buffer(tsk, cb);
if (tsk->wait_recv->sleep)
    wake_up(tsk->wait_recv);
tcp_send_control_packet(tsk, TCP_ACK);
if (tsk->wait_send->sleep)
wake_up(tsk->wait_send)
```

3.TCP重传计时器实现

将重传计时器添加到timerlist中，在扫描线程进行扫描时，根据计时器类型判断，对重传计时器进行不同的处理逻辑, 如下。

```
tsk = retranstimer_to_tcp_sock(timer);
struct send_buffer *entry;
list_for_each_entry(entry, &tsk->send_buf, list)
{
    entry->timeout -= TCP_TIMER_SCAN_INTERVAL;
    if (!entry->timeout)
    {
        if (entry->times++ == 3)
        {
            entry->times = 1;
```

```
            entry->timeout = TCP_RETRANS_INTERVAL_INITIAL;
        }
        else
        {
            char *temp = (char *)malloc(entry->len * sizeof(char));
            memcpy(temp, entry->packet, entry->len);
            ip_send_packet(temp, entry->len);
            if (entry->times == 2)
                entry->timeout = entry->times * TCP_RETRANS_INTERVAL_INITIAL;
            else
                entry->timeout = 4 * TCP_RETRANS_INTERVAL_INITIAL;
        }
    }
}
```

4.实现TCP拥塞控制状态机

拥塞控制状态的转移主要由ACK包决定，在tsk结构体中添加 ack_time 域来记录收到重复ACK包的次数。

对于收到的包 ack=snd_una 的情况，即为重复ACK，若在OPEN状态下会累积计数，当到达3时会进入RECOVERY状态，此时启动快重传以及后续的快恢复机制：将ssthresh缩小为一半，同时记录恢复点，之后启动快恢复机制，将发送队列的第一个包进行重传，后续在接受到ACK包后进行重传。在RECOVERY状态下，没收到2个ACK就使cwnd-1，实现cwnd缩小为原来的一半的机制。

```
if (cb->ack == tsk->snd_una)
{
    tsk->ack_time++;
    switch (tsk->nrstate)
    {
    case OPEN:
        if (tsk->ack_time > 2)
        {
            tsk->nrstate = RECOVER;
            tsk->ssthresh = (tsk->cwnd + 1) / 2;
            tsk->recovery_point = tsk->snd_nxt;
            struct send_buffer *entry = list_entry((&tsk->send_buf) -
                                                   > next,
                                                typeof(*entry), list);
            if (!list_empty(&tsk->send_buf))
            {
                char *temp = (char *)malloc(entry->len * sizeof(char));
                memcpy(temp, entry->packet, entry->len);
                ip_send_packet(temp, entry->len);
            }
        }
        break;
    case RECOVER:
        if (tsk->ack_time > 1)
        {
            tsk->ack_time -= 2;
            tsk->cwnd -= 1;
            if (tsk->cwnd < 1)
                tsk->cwnd = 1;
        }
        break;
    default:
```

```
        break;
    }
}
```

对于有效的ACK包，新增加cwnd加1的操作，在OPEN状态下，每确认到一个数据包，就对cwnd进行一次增大。在增大操作时，引入新的计数器，如果cwnd小于ssthresh时，实现cwnd自增1，实现慢启动机制，反之则给新计数器加1如果新计数器超过cwnd，清除新计数器的值，给cwnd+1，实现拥塞避免

```
if(tsk->nrstate == OPEN)
{
    if (tsk->cwnd < tsk->ssthresh)
        tsk->cwnd++;
    else
    {
        tsk->cnt++;
        if (tsk->cnt >= tsk->cwnd)
        {
            tsk->cnt = 0;
            tsk->cwnd++;
        }
    }
}
```

在有效ACK包情况下，首先需要检查当前ack的值是否已回到了恢复点，若到达恢复点则可以回到OPEN状态。同时还需要进行快恢复机制的处理，需要将收到的ack包对应的数据包进行重传。
在重传计时器检查的过程中，如果发生了超时重传，则需要将状态置为LOSS，记录恢复点，并将ssthresh减半，将cwnd重置为1

```
if (tsk->nrstate != LOSS)
    tsk->recovery_point = tsk->snd_nxt;
tsk->nrstate = LOSS;
tsk->ssthresh = (tsk->cwnd + 1) / 2;
tsk->cwnd = 1;
tsk->snd_wnd = MSS;
pthread_mutex_lock(&tsk->file_lock);
cwnd_dump(tsk);
pthread_mutex_unlock(&tsk->file_lock);
```

5.信息记录

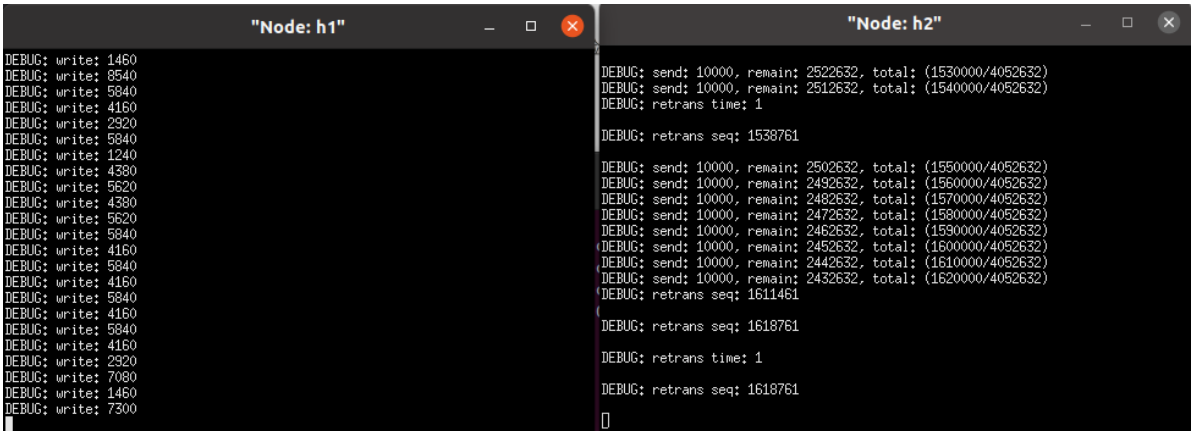
在客户端进入自己的处理程序后，会打开对应名字的文件，在每次cwnd值变化时（超时重传和收到ACK包时），执行该文件，写入对应的时间和cwnd值到文件中。同时引入文件读写锁，每次读写文件时保证是互斥访问

```
void cwnd_dump(struct tcp_sock *tsk)
{
struct timeval now;
gettimeofday(&now, NULL);
long long int duration = 1000000 * ( now.tv_sec - start.tv_sec ) + now.tv_usec -
start.tv_usec;
fprintf(record, "%lld\t%d\n", duration, tsk->cwnd);
}
```

# 实验结果

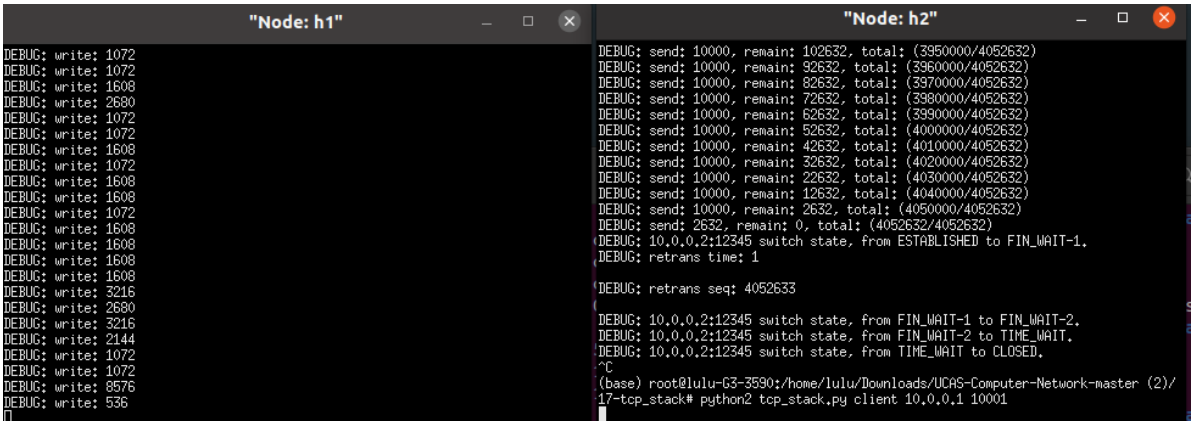
进行收发文件测试

1.h1 -> h2



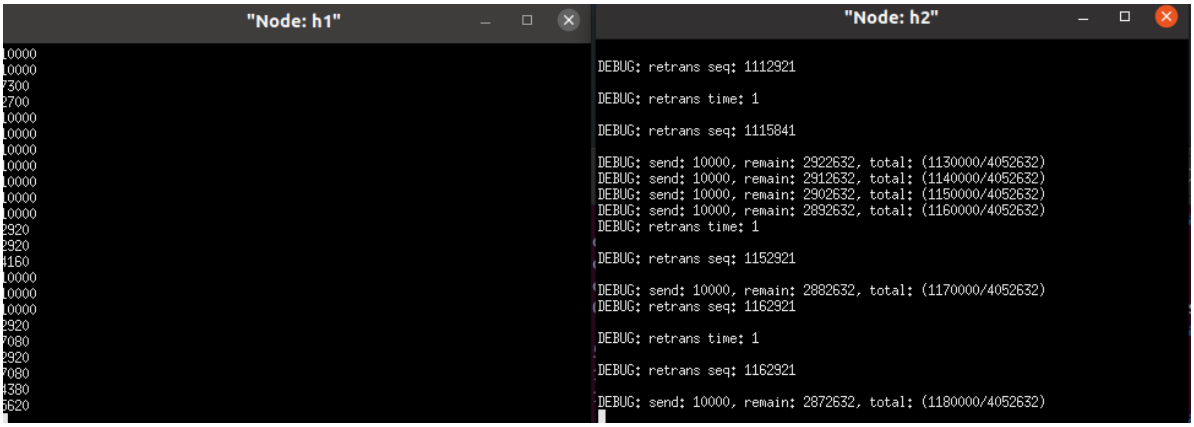

```
(base) lulu@lulu-G3-3590:~/Downloads/UCAS-Computer-Network-master (2)/17-tcp_stack$ md5sum client-input.dat server-output.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  client-input.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  server-output.dat
```

2.h1 -> Python




```
(base) lulu@lulu-G3-3590:~/Downloads/UCAS-Computer-Network-master (2)/17-tcp_stack$ md5sum client-input.dat server-output.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  client-input.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  server-output.dat
```

3.Python-> h1




```
(base) lulu@lulu-G3-3590:~/Downloads/UCAS-Computer-Network-master (2)/17-tcp_stack$ md5sum client-input.dat server-output.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  client-input.dat
48f6d51c728bd6fd31ffa3b3ff0eeafb  server-output.dat
```

# 实验总结

本次实验较为复杂，由于存在loss的情况，有时实验出现的bug难以复现，同时暴露了过去代码的一些问题。之后对TCP NewReno拥塞控制机制进行了较为完整的实现，，主要实现了拥塞控制状态的迁移以及窗口的控制，这部分实验关于cwnd的实现曾经在数据包队列实验中有所提及，当时在思考题中调研了一些拥塞控制机制，并且也对cwnd的变化趋势进行了分析。