

- 网络传输机制实验（四）
  - 实验内容
  - 实验步骤
  - 设计思路
    - New Reno方法拥塞控制状态机的实现
      - 若收到新的ACK
      - 若收到重复ACK
    - 拥塞控制
      - TCP拥塞窗口增大
      - TCP拥塞窗口减小
      - TCP拥塞窗口不变
      - 修改发送函数已经更新窗口
    - 统计CWND的变化
    - 修复之间实验的问题
      - 问题一
      - 问题二
  - 结果验证

# 网络传输机制实验（四）

2021年1月11日

蔡润泽

本实验 [Github](#) [地址](#)

## 实验内容

- TCP拥塞控制机制
  - TCP拥塞控制状态迁移
  - TCP拥塞控制机制
    - 数据包发送
    - 拥塞窗口调整
    - 重传数据包
- TCP拥塞控制机制实现

## 实验步骤

- 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.py)

- 在节点h1上执行TCP程序
  - 执行脚本(disable\_tcp\_rst.sh, disable\_offloading.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\_tcp\_rst.sh, disable\_offloading.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

## 设计思路

### New Reno方法拥塞控制状态机的实现

根据TCP拥塞控制状态迁移图, 设计状态机的跳转。

TCP初始状态为 `OPEN`。

#### 若收到新的ACK

- 当TCP处于 `DISORDER` 状态时, 则跳转回初始状态 `OPEN`。
- 当TCP处在 `LOSS` 状态时, 若收到的ACK大于切换到 `LOSS` 状态时的snd\_nxt时, 跳转回OPEN。
- 当TCP处在 `RECOVERY` 状态时, 若收到的ACK大于切换到 `RECOVERY` 状态时的snd\_nxt时, 跳转回OPEN。

#### 若收到重复ACK

- 当TCP处在 `OPEN` 状态下, 收到重复ACK时, `tsk->dupacks` 加1, 状态切换到 `DISORDER`。
- 当TCP处在 `OPEN` 状态下, 收到重复ACK时, `tsk->dupacks` 加1, 当 `tsk->dupacks = 3` 时, 状态切换到 `RECOVERY`。

## 拥塞控制

### TCP拥塞窗口增大

- 慢启动 (Slow Start)
  - 对方每确认一个报文段, cwnd增加1MSS, 直到cwnd超过sssthresh值
  - 经过1个RTT, 前一个cwnd的所有数据被确认后, cwnd大小翻倍
- 拥塞避免 (Congestion Avoidance)
  - 对方每确认一个报文段, cwnd增加 $(1 \text{ MSS}) / \text{CWND} * 1 \text{ MSS}$
  - 经过1个RTT, 前一个cwnd的所有数据被确认后, cwnd增加1 MSS

具体实现如下：

```
void update_cwnd(struct tcp_sock *tsk) {
    if ((int)tsk->cwnd < tsk->sssthresh) {
        tsk->cwnd ++;
    } else {
        tsk->cwnd += 1.0/tsk->cwnd;
    }
}
```

注：TSK为实现方便，将cwnd的变量类型设为了 `float` 。

## TCP拥塞窗口减小

- 快重传（Fast Retransmission）
  - ssthresh减小为当前cwnd的一半： $\text{ssthresh} \leftarrow \text{cwnd} / 2$
  - 新拥塞窗口值  $\text{cwnd} \leftarrow \text{新的ssthresh}$
- 超时重传（Retransmission Timeout）
  - Ssthresh减小为当前cwnd的一半： $\text{ssthresh} \leftarrow \text{cwnd} / 2$
  - 拥塞窗口值cwnd减为1 MSS

在具体实现减半操作时时，每收到一个ACK，cwnd的值减少 `0.5` 。

## TCP拥塞窗口不变

- 快恢复（Fast Recovery）
  - 进入：在快重传之后立即进入
  - 退出：
    - 当对方确认了进入FR前发送的所有数据时，进入Open状态
    - 当触发RTO后，进入Loss状态
  - 在FR内，收到一个ACK：
    - 若该ACK没有确认新数据，则说明inflight减一，cwnd允许发送一个新数据包
    - 若该ACK确认了新数据
    - 如果是Partial ACK，则重传对应的数据包
    - 如果是Full ACK，则退出FR阶段

具体实现如下：

```

if (tsk->nr_state == TCP_RECOVERY) {
    if (isNewAck) {
        if (tsk->cwnd > tsk->ssthresh && tsk->cwnd_flag == 0) {
            tsk->cwnd -= 0.5;
        } else {
            tsk->cwnd_flag = 1;
        }
        if (cb->ack < tsk->recovery_point) {
            retrans_send_buffer_packet(tsk);
        } else {
            tsk->nr_state = TCP_OPEN;
            tsk->dupacks = 0;
        }
    } else {
        tsk->dupacks ++;
        if (tsk->cwnd > tsk->ssthresh && tsk->cwnd_flag == 0) {
            tsk->cwnd -= 0.5;
        } else {
            tsk->cwnd_flag = 1;
        }
    }
}
}

```

上述函数中有标志位 `cwnd_flag`，当flag为 `0` 时说明此时cwnd还需要继续下降。

## 修改发送函数已经更新窗口

发送数据包时，需要根据发送窗口大小进行判断，本设计中引入了函数，设计如下：

```

int is_allow_to_send (struct tcp_sock *tsk) {
    int inflight = (tsk->snd_next - tsk->snd_una)/MSS - tsk->dupacks;
    return max(tsk->snd_wnd / MSS - inflight, 0);
}

```

若该函数的返回值为 `0`，则 `sleep_on(tsk->wait_send);`

另外对于 `tcp_update_window` 进行了如下修改：

```

static inline void tcp_update_window(struct tcp_sock *tsk, struct tcp_cb *cb) {
    u16 old_snd_wnd = tsk->snd_wnd;
    tsk->snd_wnd = min(cb->rwnd, tsk->cwnd * MSS);
    if ((int)old_snd_wnd <= 0) {
        wake_up(tsk->wait_send);
    }
}

```

（注：若tsk->adv\_wnd之前收到ACK时被赋成cb->rwnd的值，此处的cb->rwnd被替换成tsk->adv\_wnd也行）

# 统计CWND的变化

PPT中建议每当值发生改变时，就记录CWND的变化，但是这样需要在TSK中新增两个变量用来记录时间和FILE的句柄。因此在本设计中改为每间隔一段时间，就扫描当前CWND的值。另外需要注意的是，由于本实验中存在两个tsk，因此统计值的时候需要指定统计客户端的cwnd即tsk->parent等于NULL的tsk的CWND。

另外数据处理用了ipynb实现。

## 修复之间实验的问题

### 问题一

补充了流量控制，在之前的设计中，忘记改变tsk->rwnd的值，让他一直保持在初始默认状态，新设计中将其改为了，当写ring\_buffer时，将ring\_buffer的剩余空间赋值给tsk->rwnd。

### 问题二

之前实验三中传输时间较慢，我以为是因为没有实现快重传。在本次实验实现了快重传后，发现TCP依旧总是需要超时重传，查看代码发现，之前retrans定时器set和update有问题，其time\_out并没有赋值时，而是在编辑器自动补全时，变成了给time\_wait的timeout赋值。这个问题在本次代码设计中被修复。

## 结果验证

本次实验传输的数据大小为 1,350,880B ，设置的丢包率为 5% 。

本次实验的传输结果如下：

```
"Node: h1"
state: FIN_WAIT-1
rcv_nxt:1350881, ack:846930887, seq:1350881, len:0
DEBUG: 10.0.0.1:10001 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.1:10001 switch state, from FIN_WAIT-2 to TIME_WAIT.
TODO: implement tcp_set_timewait_timer here.
flags: 0x10
state: TIME_WAIT
rcv_nxt:1350882, ack:846930888, seq:1350882, len:0
retrans: time 0
retrans seq: 846930887
retrans: time 1
retrans seq: 846930887
flags: 0x11
state: TIME_WAIT
rcv_nxt:1350882, ack:846930888, seq:1350881, len:0
DEBUG: 10.0.0.1:10001 switch state, from TIME_WAIT to CLOSED.
TODO: implement free_tcp_sock here.
^C
root@ubuntu:~/code/18-tcp_stack# diff client-input.dat server-output.dat
root@ubuntu:~/code/18-tcp_stack# md5sum client-input.dat
320bd9ed397d7820797a0c55a0215a63  client-input.dat
root@ubuntu:~/code/18-tcp_stack# md5sum server-output.dat
320bd9ed397d7820797a0c55a0215a63  server-output.dat
root@ubuntu:~/code/18-tcp_stack#

"Node: h2"
***** send FIN *****
flags: 0x11
state: FIN_WAIT-1
rcv_nxt:846930887, ack:1350881, seq:846930887, len:0
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.
TODO: implement tcp_set_timewait_timer here.
flags: 0x10
state: TIME_WAIT
rcv_nxt:846930888, ack:1350882, seq:846930888, len:0
retrans: time 0
retrans seq: 1350881
flags: 0x11
state: TIME_WAIT
rcv_nxt:846930888, ack:1350882, seq:846930887, len:0
retrans: time 1
retrans seq: 1350881
flags: 0x11
state: TIME_WAIT
rcv_nxt:846930888, ack:1350882, seq:846930887, len:0
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
TODO: implement free_tcp_sock here.

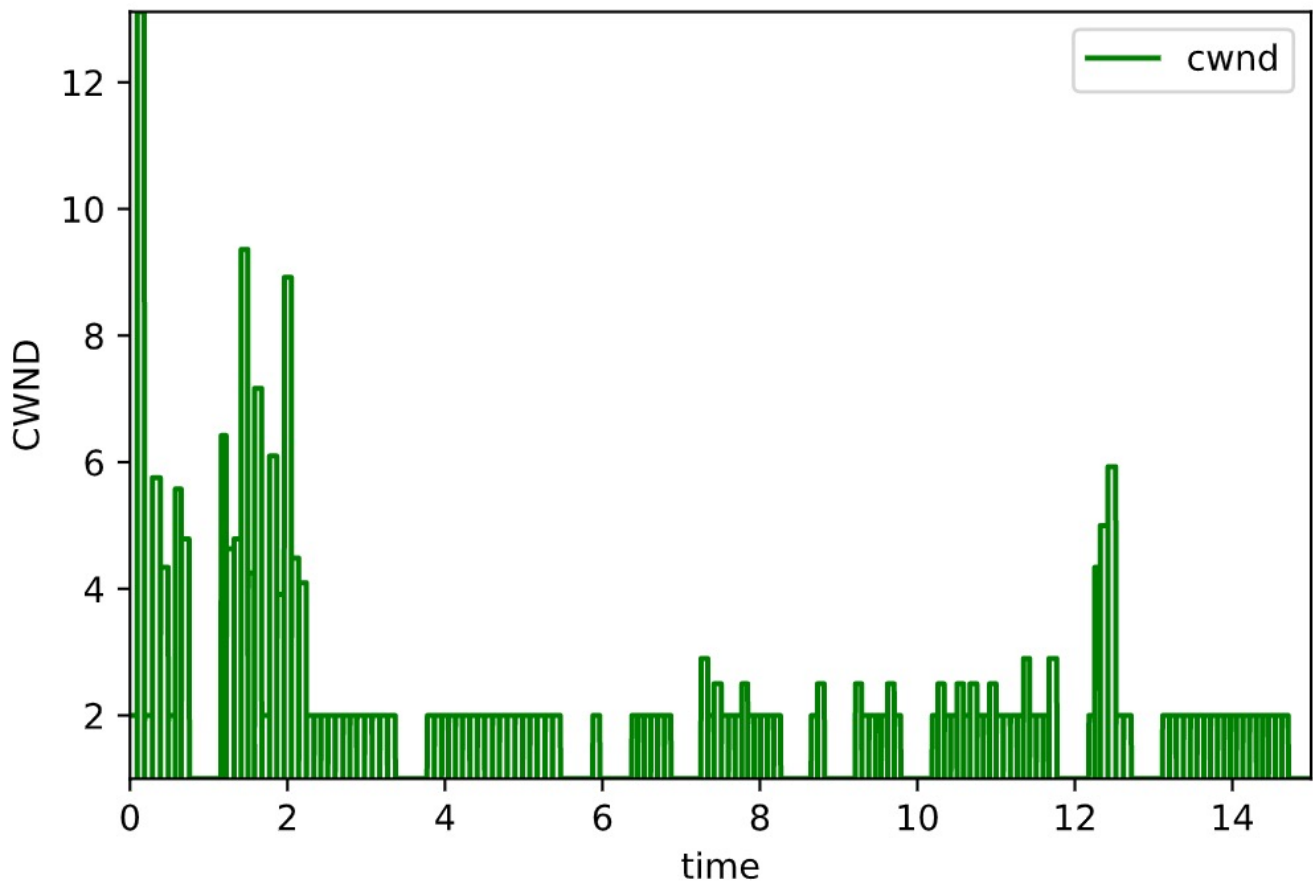
vincent@ubuntu:~/code/18-tcp_stack
vincent@ubuntu:~$ cd code/18-tcp_stack/
vincent@ubuntu:~/code/18-tcp_stack$ make
gcc -c -g -Wall -Iinclude tcp_in.c -o tcp_in.o
gcc -c -g -Wall -Iinclude tcp_sock.c -o tcp_sock.o
gcc arp.o arpcache.o icmp.o ip.o main.o packet.o rtable.o rtable_internal.o tcp
.o tcp_apps.o tcp_in.o tcp_out.o tcp_sock.o tcp_timer.o -o tcp_stack -lpthread
vincent@ubuntu:~/code/18-tcp_stack$ sudo python tcp_topo_loss.py
[sudo] password for vincent:
*** Error: RTNETLINK answers: No such file or directory
*** Error: RTNETLINK answers: No such file or directory
mininet> xterm h1 h2
mininet>
```



上图可知，可知本次实验结果符合预期，客户端发送的文件与服务器端接受的文件一致。

CWND的变化曲线如下：

# CWND



与预期结果存在两点小的差异，原因如下：

- 不够平滑是因为本设计中改为每间隔一段时间，就扫描当前CWND的值，而扫描的时间间隔相比于CWND的变化稍大。
- 由于为测试该设计的鲁棒性，本实验环境的丢包率设置为5%，这一数字相对较大，导致需要超时重传的次数也相对增多，最终导致很多时间下CWND的大小接近于 1 。