计算机网络实验四报告

1 代码设计

1.1 修改思路

我实现的是调整过的四个状态的状态机，主要是增加了 void tcp_ congestion_ control(struct tcp_ sock *tsk, struct tcp_ cb *cb, int ack_ valid) 这个函数，并在 tcp_ process 的 client 端收到 ACK 的处理逻辑添加该函数，修改 CWND 等参数的初始值。
对于记录和画图，我选择用每个 ACK 进行记录，而不是用新线程固定时间记录（因此图和时间的捕捉不太完全一致，后面解释原因），再用一个 python 脚本进行画图。由于 ACK 很多，一张图太密集，我选择分段绘制。记录 ACK 和拥塞控制函数在一起，每次调用该函数都记录，这样保证了不会漏掉状态和 ACK。

1.2 主要修改代码

1.tcp_ sock.h 的修改首先，将 cwnd 由 u32 类型改为 float 类型，使其更精确。
然后加入拥塞控制状态并在 struct tcp_ sock 里加入 int cstate 指示状态。

```
// 拥塞控制状态
enum tcp_congestion_state {
    TCP_CONG_OPEN,        // 正常状态（包含慢启动和拥塞避免）
    TCP_CONG_DISORDER,    // 失序状态
    TCP_CONG_RECOVERY,    // 快速恢复状态
    TCP_CONG_LOSS        // 超时状态
    };
```

2. 拥塞控制函数 (带注释)

```
void tcp_congestion_control(struct tcp_sock *tsk, struct tcp_cb *cb, int
    ack_valid)
{
    if(!ack_valid) return;
    log(DEBUG,"tsk->c_state=%d",tsk->c_state);
    static FILE *fp = NULL;
    static int time_us = 0;

    if (!fp) {
        fp = fopen("cwnd.txt", "w");
        if (!fp) {
            log(ERROR, "Failed to open cwnd.txt");
            return;
        }
    }
```

```
if (less_or_equal_32b(tsk->snd_una, cb->ack) && less_or_equal_32b(cb->ack,
    tsk->snd_nxt)){
        // 记录当前拥塞控制参数
        time_us += 1000;  // 假设每个ACK间隔1ms
        fprintf(fp, "%d␣%f␣%u␣%u␣%d\n", time_us, tsk->cwnd, tsk->ssthresh, tsk
            ->adv_wnd,tsk->c_state);
        fflush(fp);
       }
 // log(DEBUG,"cb->ack=%u snd_una=%u",cb->ack,tsk->snd_una);
  switch (tsk->c_state) {
      case TCP_CONG_OPEN:
          if (cb->ack > tsk->snd_una) {
              // 收到新的ACK
              tsk->dup_ack_count = 0;
              if (tsk->cwnd < tsk->ssthresh) {
                  // 慢启动阶段,指数增长
                  log(DEBUG,"slow␣start");
                  tsk->cwnd+=(float)(1*TCP_MSS);
              } else {
                  // 拥塞避免阶段,线性增长
                  log(DEBUG,"congestion␣avoidance");
                  tsk->cwnd += (float)((float)(TCP_MSS*TCP_MSS)/(float)tsk->
                      cwnd);
                 // tsk->cwnd+=(float) (1.0*TCP_MSS)/(float)tsk->cwnd;
              }
          }
          else if(cb->ack==tsk->snd_una){
              // 收到重复ACK
              tsk->dup_ack_count++;
              tsk->c_state = TCP_CONG_DISORDER;
              if (tsk->dup_ack_count == 3) {
                  // 进入快速恢复
                  tsk->c_state = TCP_CONG_RECOVERY;
                  tsk->ssthresh = tsk->cwnd / 2;
                  tsk->cwnd = (float)(tsk->ssthresh + 3*TCP_MSS);
                  tsk->recovery_point = tsk->snd_nxt;
                  tcp_retrans_send_buffer(tsk);
              }
          }
          break;

      case TCP_CONG_DISORDER:
          if (cb->ack > tsk->recovery_point) {
              // 新数据被确认,返回OPEN状态
              tsk->c_state = TCP_CONG_OPEN;
              tsk->dup_ack_count = 0;
              tsk->recovery_point = tsk->snd_nxt;
          } else if (cb->ack == tsk->snd_una) {
              // 继续收到重复ACK
```

```
                tsk->dup_ack_count++;
                if (tsk->dup_ack_count == 3) {
                    tsk->c_state = TCP_CONG_RECOVERY;
                    tsk->ssthresh = tsk->cwnd / 2;
                    tsk->cwnd = (float)(tsk->ssthresh + 3*TCP_MSS);
                    tcp_retrans_send_buffer(tsk);
                }
            }
            break;

        case TCP_CONG_RECOVERY:
            if (cb->ack > tsk->recovery_point) {
                // 新数据被确认,返回OPEN状态
                tsk->c_state = TCP_CONG_OPEN;
                tsk->cwnd = (float)tsk->ssthresh;
                tsk->dup_ack_count = 0;
                tsk->recovery_point = tsk->snd_nxt;
            } else if (cb->ack == tsk->snd_una) {
                // 继续收到重复ACK
                tsk->cwnd+=(float)(TCP_MSS);
            }
            break;

        case TCP_CONG_LOSS:
            // 从超时中恢复
            tsk->c_state = TCP_CONG_OPEN;
            tsk->dup_ack_count = 0;
            tsk->ssthresh = tsk->cwnd / 2;
            tsk->cwnd = (float)TCP_MSS;
            tsk->recovery_point = tsk->snd_nxt;
            break;
    }
    log(DEBUG,"end");
}
```

3.tcp_ process 的主要修改

```
case TCP_ESTABLISHED:
           // log(DEBUG,"ESTABLISHED1");
            if (cb->flags & TCP_FIN) {//server收到FIN包
              //  printf("ESTABLISHEDFIN");
                tsk->rcv_nxt = cb->seq_end;
                tcp_set_state(tsk, TCP_CLOSE_WAIT);
                if (ring_buffer_empty(tsk->rcv_buf)) {
                    // 没有待发送数据，直接发送 FIN 包，进入 TCP_LAST_ACK 状态
                    tcp_send_control_packet(tsk, TCP_ACK | TCP_FIN);
                    tcp_set_state(tsk, TCP_LAST_ACK);}
                    else {
```

```
                    // 有待发送数据，先发送 ACK 包，等待数据传输完毕再发送
                       FIN 包
                    tcp_send_control_packet(tsk, TCP_ACK);
                    sleep_on(tsk->wait_send);//to be changed
                    tcp_send_control_packet(tsk, TCP_FIN);
                    tcp_set_state(tsk, TCP_LAST_ACK);
                }
            wake_up(tsk->wait_recv);
        } else if ((cb->flags & TCP_ACK)&&!tsk->parent) {
           //  printf("established_ack\n");
            //tsk->snd_una = cb->ack;

            int ack_valid = 1;


            if (less_than_32b(cb->ack, tsk->snd_una)) {
                // 收到的ACK确认的序号小于已确认序号，可能是由于重传导致的
                   重复ACK
                log(DEBUG,"ack_valid=0");
                ack_valid = 0;
            }

           tcp_congestion_control(tsk, cb, ack_valid);
            tcp_update_send_buffer(tsk, cb->ack);
            tcp_update_retrans_timer(tsk);
            tcp_update_window_safe(tsk, cb);

            wake_up(tsk->wait_send);
        }
        break;
```

在 client 端在 established 状态收到 ACK 时判断是否合法并且调用拥塞控制函数。
并且在 client 端进入 established 状态时初始化参数：

```
tsk->cwnd=(float)TCP_MSS;
tsk->ssthresh=(u32)64 *1024;
tsk->dup_ack_count=0;
tsk->c_state=TCP_CONG_OPEN;
```

4. 画图假设每个 ACK 间隔 1ms, 我们分段每 1000ms 用 py 脚本作图。
作图：

```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams.update({'font.size': 14})

data = np.loadtxt('cwnd.txt')
```

```
time = data[:, 0] / 1000
cwnd = data[:, 1]
ssthresh = data[:, 2]
state = data[:, 4].astype(int)

interval = 1000
num_segments = int(np.ceil(time[-1] / interval))

# 修改颜色方案，使LOSS状态更明显
state_colors = {
    0: 'green',      # OPEN
    1: 'yellow',     # DISORDER
    2: 'red',        # RECOVERY
    3: 'black'       # LOSS - 使用黑色
}

state_names = ["OPEN", "DISORDER", "RECOVERY", "LOSS"]
used_states = set()

for i in range(num_segments):
    # ...existing code...
    plt.figure(figsize=(12, 8))
    used_states.clear()

    # 绘制ssthresh
    s = ssthresh[i * interval:(i + 1) * interval]  # Define 's' as a segment of
        'ssthresh'
    st = state[i * interval:(i + 1) * interval]  # Define 'st' as a segment of
       'state'
    t = time[i * interval:(i + 1) * interval]   # Define 't' as a segment of '
       time'
    c = cwnd[i * interval:(i + 1) * interval]   # Define 'c' as a segment of '
       cwnd'
    plt.plot(t, s, 'r--', label='ssthresh', linewidth=2)

    # 标记LOSS状态发生的位置
    for j in range(1, len(st)):
        if st[j] == 3:  # LOSS状态
            plt.axvline(x=t[j], color='black', linestyle=':', alpha=0.5)
            plt.axvspan(t[j], t[j+1] if j+1 < len(t) else t[-1],
                        color='gray', alpha=0.2)

    # 为每个状态绘制一段折线
    last_idx = 0
    last_state = st[0]
    for j in range(1, len(st)):
        if st[j] != last_state:
            label = f'cwnd␣({state_names[int(last_state)]})' if last_state not
                in used_states else None
```

```
            if last_state == 3:  # LOSS状态使用更粗的线
                linewidth = 3
            else:
                linewidth = 2
            plt.plot(t[last_idx:j+1], c[last_idx:j+1],
                     color=state_colors[last_state],
                     linewidth=linewidth,
                     label=label)
            used_states.add(last_state)
            last_idx = j
            last_state = st[j]

    # 绘制最后一段
    label = f'cwnd ({state_names[int(last_state)]})' if last_state not in
        used_states else None
    linewidth = 3 if last_state == 3 else 2
    plt.plot(t[last_idx:], c[last_idx:],
             color=state_colors[last_state],
             linewidth=linewidth,
             label=label)

    start_time = i * interval
    end_time = (i + 1) * interval if (i + 1) * interval <= time[-1] else int(
        time[-1])
    plt.title(f'TCP Congestion Control ({start_time}-{end_time}ms)', fontsize
        =16)
    plt.xlabel('Time (ms)', fontsize=14)
    plt.ylabel('Window Size (bytes)', fontsize=14)

    plt.grid(True, alpha=0.3)
    plt.legend(fontsize=12)
    plt.tight_layout()

    plt.savefig(f'segment_{i+1}.png', dpi=300, bbox_inches='tight')
    plt.close()
```

1.3 实验结果分析

在一次完整的传输过程里, 共获取了约 4000 份 ACK, 做成了四个阶段的图片。文件传输没有丢失数据。

图中用不同颜色标注了 ssthresh 的变化, OPEN (慢启动和拥塞避免状态), Disorder (<3dupACK) ,RECOVERY(快重传),LOSS(超时重传) 四个部分的出现。

```
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
guan@guan-ThinkPad-Ubuntu ~/D/计/n/exp3 (main)> diff client-input.dat server-output.dat
guan@guan-ThinkPad-Ubuntu ~/D/计/n/exp3 (main)>
```

图 1: no difference

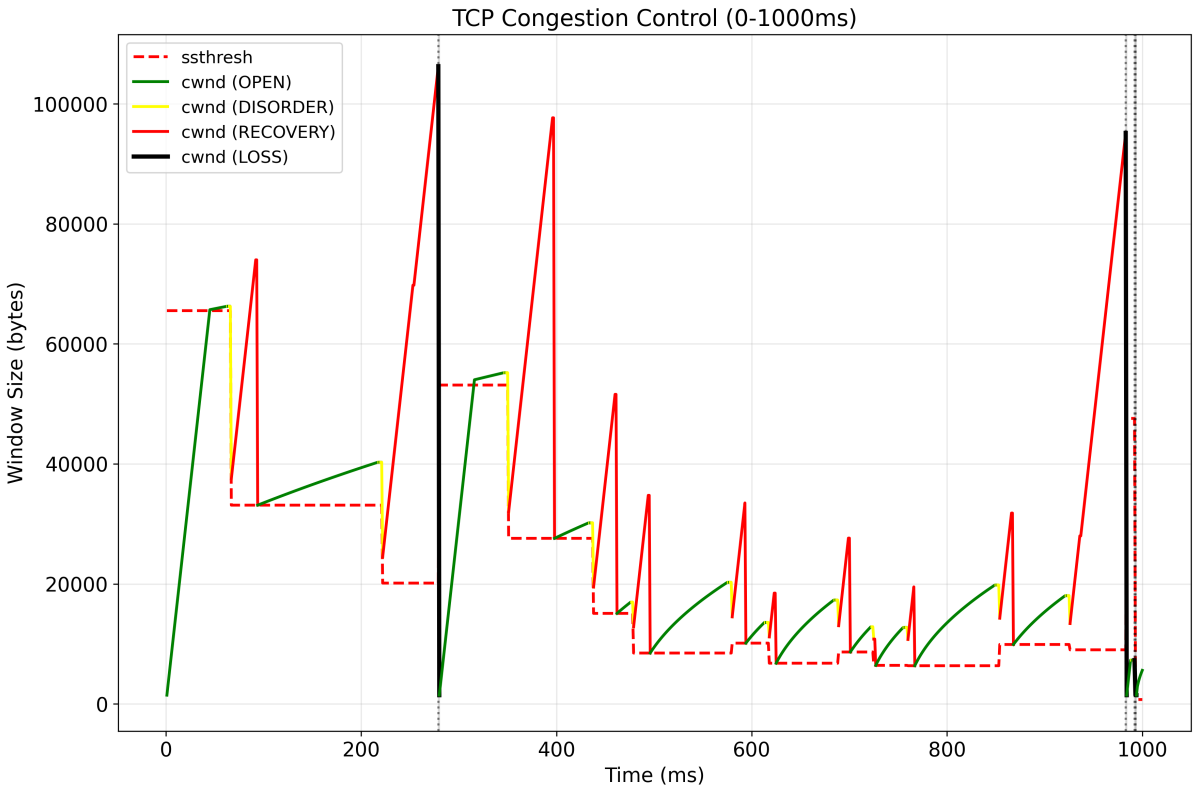

图 2: 拥塞控制记录图 1

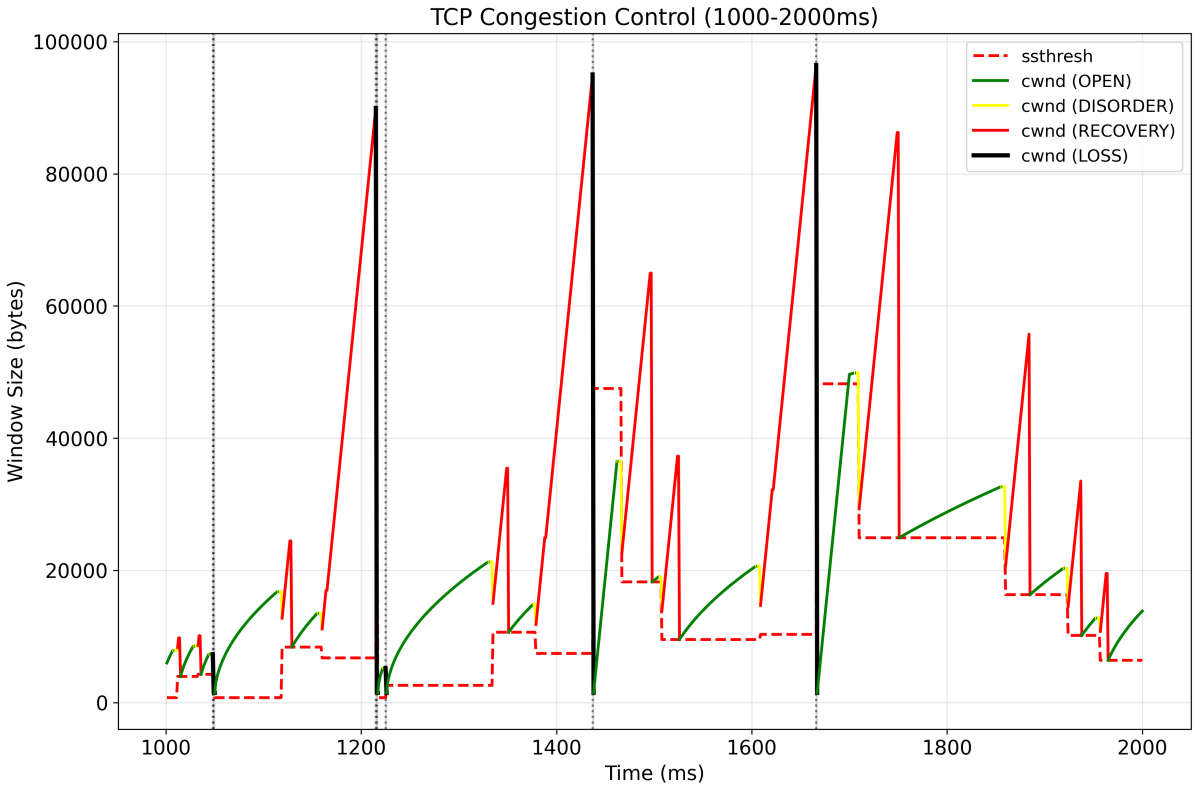
TCP Congestion Control (1000-2000ms)
ssthresh
cwnd (OPEN)
cwnd (DISORDER)
cwnd (RECOVERY)
cwnd (LOSS)
Window Size (bytes)
Time (ms)
0
20000
40000
60000
80000
100000
1000
1200
1400
1600
1800
2000

图 3: 2

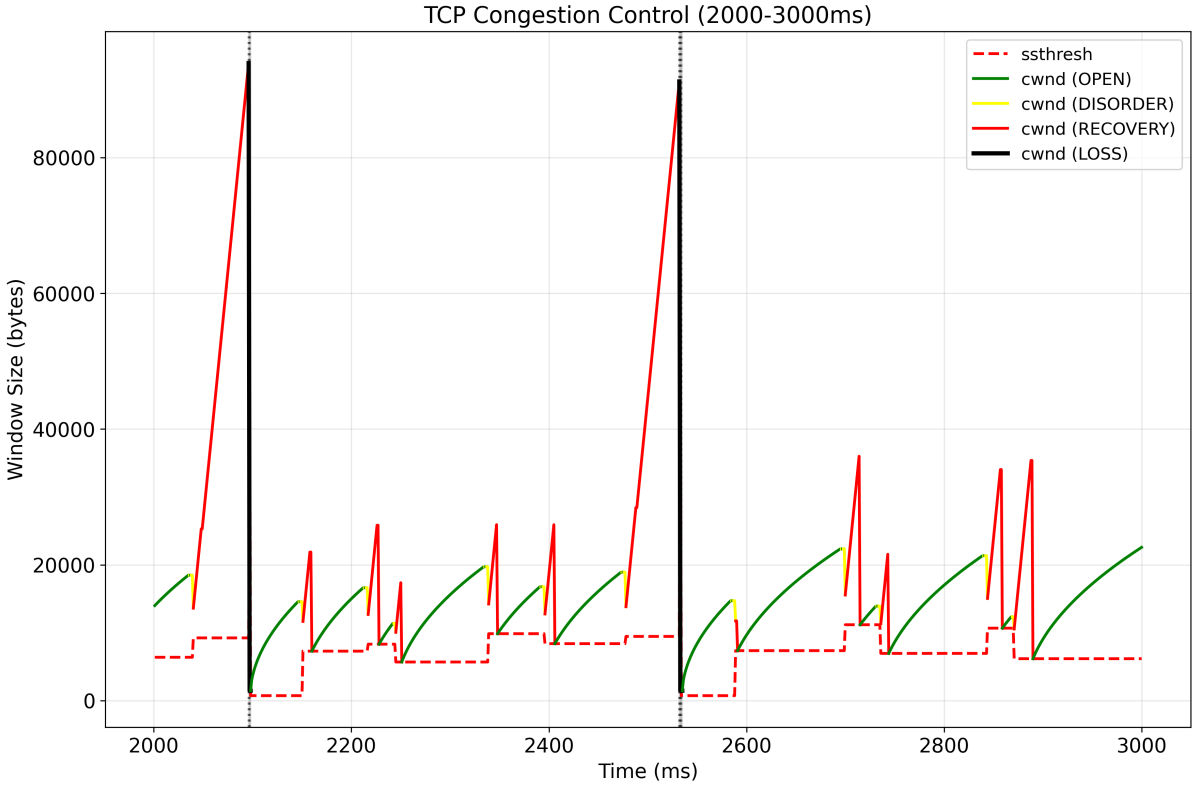
TCP Congestion Control (2000-3000ms)
ssthresh
cwnd (OPEN)
cwnd (DISORDER)
cwnd (RECOVERY)
cwnd (LOSS)
Window Size (bytes)
Time (ms)
0
20000
40000
60000
80000
2000
2200
2400
2600
2800
3000

图 4: 3

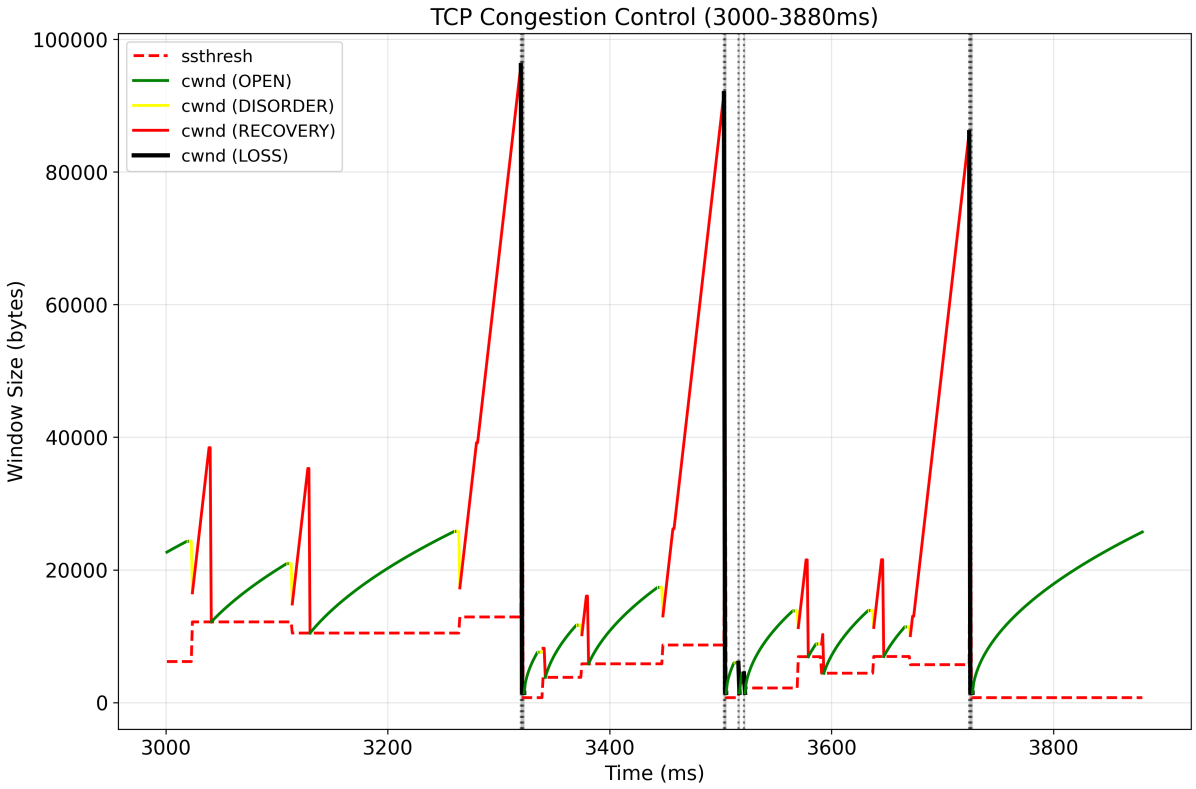

图 5: 4

很容易可以看出黄色对应了拥塞窗口减半的情况。

这里为什么慢启动是一个直线而拥塞控制反而是曲线呢? 因为我是每个 ACK 进行捕捉的, 而不是每隔一段时间, 对 ACK 来说慢启动就是每次加 1MSS, 因此是直线, 而拥塞控制对于 ACK 来说由于 CWND 在增加增加的会相对越来越慢, 但是对于时间来说基本是线性的。

从图上可以看出, 在有丢包的情况下, 拥塞控制如果没有超时重传, 就在慢启动(可能到拥塞控制)-快速重传-慢启动的循环中, 这是符合 new reno 的。

ssthresh 一直在根据 CWND 的值进行更新, 并一般比 CWND 的值低(除了一开始慢启动显著高, 后续拥塞控制阶段和快速重传均比 CWND 低), 这也是符合 new reno 的。

可以看出重传出现了一个大峰, 这是因为快速重传迅速拔高了 CWND 然后传完又恢复 1MSS 开始重新慢启动导致的, 符合 new reno。重传的峰有时候很高是因为有很多的 dupACK, 每个 dupACK 都在 recovery 状态加了一个 MSS, 导致某些时候峰很高, 但是有的时候快速重传没来超时先到, 那么 CWND 就会被重置重新进行慢启动。

慢启动的指数函数比拥塞控制的直线更陡峭。

有的时候一开始慢启动并没有达到 ssthresh 进入拥塞控制, 是因为丢包的 dupACK 来得太快, 而 ssthresh 的初始值很大(64kb), 直接进入重传了。(这张图一开始的拥塞控制阶段也很短)

可以看出 ssthresh 有增加有减少。

1.4 其他小修改

用 tsk->parent 做了是 client 端还是 server 端的判断,防止 probe 包(keep-alive 包)影响逻辑。将错误调用 tcp _ update _ window 的 server 端的部分去除了。