计算机网络实验 14 报告

学号　2017K8009929059
姓名　於修远

网络传输机制实验四

一、实验内容

- 补全实验代码，实现 TCP 拥塞状态机的状态维护和更新，并实现各拥塞状态下的控制机制。
- 运行给定网络拓扑，用两个节点分别作为服务器节点和客户端节点，使 TCP 传输可以在有丢包的链路环境中正确运行，并通过二维坐标图展示拥塞窗口 cwnd 的变化情况。

二、实验流程

（一）拥塞状态机切换和拥塞窗口控制

拥塞状态机的切换是本次实验的核心，拥塞状态机的控制几乎完全由带 ACK 标志位的数据包来完成。在本设计中，以 MSS 作为 cwnd 的单位，在计算发送窗口等时乘 MSS 值即可使用，降低维护复杂度。根据 ack 字段的值，可以将接收到的 ACK 消息分为三类：无效 ACK，重复 ACK，有效 ACK。第一类消息的 ack 字段小于接收方的` snd_una `，应当直接丢弃，否则会引发发送方拥塞窗口的异常减小。第二类消息的 ack 字段等于接收方当前的` snd_una `，在拥塞状态为 OPEN 时会累计计数，达到 3 时触发切换到 RCVR 状态（快重传及快恢复）的逻辑；在拥塞状态为 RCVR 时每接收到两个重复 ACK 就会使 cwnd 值减一，从而实现 cwnd 值递减至一半的效果。值得一提的是，应当在更新 cwnd 后进行一次检查，对小于 1 的 cwnd 值应当恢复至 1，防止在极端情况下负数 cwnd 值的出现。

```
if (cb->ack == tsk->snd_una) {
    tsk->rep_ack++;
    switch (tsk->cgt_state) {
        case OPEN:
            if (tsk->rep_ack > 2) {...} else break;
        case RCVR:
            if (tsk->rep_ack > 1) {
                tsk->rep_ack -= 2;
                tsk->cwnd -= 1;
                if (tsk->cwnd < 1) tsk->cwnd = 1;
            }
            break;
        default:
            break;
    }
```

如果触发了从 OPEN 状态向 RCVR 状态的切换，具体而言，还应当更新 ssthresh 的值为当前 cwnd 的一半，以当前` snd_nxt `的值记录恢复点，至此完成了快重传的维护工作。同时，需要进行第一次快恢复（后续的快恢复根据新的 ACK 消息触发，只有本次是主动发送），即将发送队列中的第一个数据包进行一次重传。只要运行正常，触发快重传时发送队列应当必然非空，所以该逻辑不会引起内存错误。

```
if (tsk->rep_ack > 2) {
    tsk->cgt_state = RCVR;
    tsk->ssthresh = (tsk->cwnd + 1) / 2;
    tsk->recovery_point = tsk->snd_nxt;
    struct send_buffer *buf = NULL;
    list_for_each_entry(buf,  head: &tsk->send_buf, list) break;
    if (!list_empty( list: &tsk->send_buf)) {
        char *temp = (char *) malloc( size: buf->len * sizeof(char));
        memcpy(temp, buf->packet, buf->len);
        ip_send_packet(temp, buf->len);
    }
} else break;
```

对于有效 ACK 消息，沿用上一次实验中逐个检查发送队列并清除已确认数据包的逻辑。一处改动是每确认一个数据包，要对窗口进行一次增大：若当前 cwnd 不大于门限值 ssthresh，直接给 cwnd 加一，一个 RTT 后即可实现倍增，也就是慢启动机制；否则给计数器` cwnd_unit `加一，并在其不小于 cwnd 时将其清空，给 cwnd 加一，一个无丢包 RTT 后即可实现 cwnd 自增 1 的行为，也就是拥塞避免机制。此外，窗口增大仅在拥塞状态为 OPEN 时进行，否则可能导致窗口的突发性增长。

```
static inline void tcp_cwnd_inc(struct tcp_sock *tsk) {
    if (tsk->cgt_state != OPEN) return;
    if (tsk->cwnd < tsk->ssthresh) {
        tsk->cwnd += 1;
    }
    else {
        tsk->cwnd_unit += 1;
        if (tsk->cwnd_unit >= tsk->cwnd) {
            tsk->cwnd_unit = 0;
            tsk->cwnd += 1;
        }
    }
}
```

在更新发送队列后，还需要进行状态机的检查，因为对于非 OPEN 状态而言，接收到有效 ACK 消息意味着有数据包被重传成功。此时，不论是何种重传情况，共通的逻辑是检查新 ack 字段是否不小于恢复点的记录值。若大于，意味着之前发送队列中所有以缓存待重传或确认的包已全部被接收到，已完成了重传的恢复工作，所以可以将拥塞状态机切换回到 OPEN 状态。

对于快恢复的情况，还需要加入重传 ack 对应的数据包，也就是更新后的发送队列首数据包的逻辑。因为快恢复认为链路只丢失了少量数据包，所以通过发送对应的特定数据包来复原。考虑到前述清除发送队列已确认缓存的逻辑，跳出遍历时的缓存指针恰好是缓存队列的新首位，所以直接使用即可。

```
switch (tsk->cgt_state) {
    case RCVR:
        if (less_or_equal_32b( a: tsk->recovery_point,  b: cb->ack)) {
            tsk->cgt_state = OPEN;
        } else {
            char *temp = (char *) malloc( size: buf->len * sizeof(char));
            memcpy(temp, buf->packet, buf->len);
            ip_send_packet(temp, buf->len);
        }
        break;
    case LOSS:
        if (less_or_equal_32b( a: tsk->recovery_point,  b: cb->ack)) {
            tsk->cgt_state = OPEN;
        }
        break;
    default:
        break;
}
```

除了上述根据 ACK 消息完成的拥塞状态机控制逻辑以外，仅有超时重传会影响拥塞状态。如果发送发被触发了超时重传，应当先后记录恢复点，切换拥塞状态为LOSS，减半门限值 ssthresh，把 cwnd 重置为 1，并将发送窗口减小为 MSS（相当于手动执行了一次发送窗口更新）。因为超时重传状态中不会更新拥塞窗口等大小，也不会追加发送新的数据包，所以这一设计不会影响超时重传完成后正常的数据收发。恢复点的作用与快重传、快恢复中的作用相仿，复用后便于判定重传完成。需要注意的是，仅在从非超时重传状态切换进入时需要记录恢复点，超时重传也不存在恢复点判定以外的状态跳出方式。这是为了防止超时重传状态的无限延续或与其他状态的交叉混乱。

```
if (!buf->timeout) {
    if (tsk->cgt_state != LOSS) tsk->recovery_point = tsk->snd_nxt;
    tsk->cgt_state = LOSS;
    tsk->ssthresh = (tsk->cwnd + 1) / 2;
    tsk->cwnd = 1;
    tsk->snd_wnd = MSS;
```

（二）发送窗口维护

发送窗口的维护时点是接收到重复或有效 ACK 消息，且完成上述的拥塞窗口更新之后。原有的窗口更新逻辑未考虑 cwnd，而是直接使用接收窗口大小。本次实验中引入了 cwnd，发送窗口每次都应当更新为 cwnd 和接收窗口中较小的一方。值得注意的是，由于 cwnd 使用 MSS 作为单位，所以在计算过程中应当乘上该值。

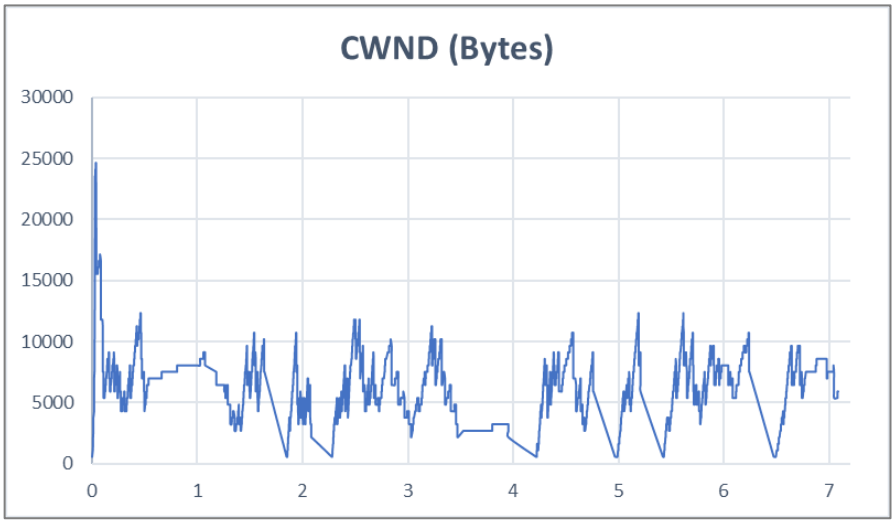
```
static inline void tcp_update_window(struct tcp_sock *tsk, struct tcp_cb *cb) {
    u16 old_snd_wnd = tsk->snd_wnd;
    tsk->adv_wnd = cb->rwnd;
    tsk->snd_wnd = min( x: tsk->adv_wnd,  y: tsk->cwnd * (MSS));
    if (old_snd_wnd == 0)
        wake_up(tsk->wait_send);
}
```

（三）拥塞窗口变化记录

本项目中，提供了两种记录拥塞窗口变化情况的方法，通过`tcp_socket.h`头文件中的宏定义`LOG_BY_TIME`来实现切换。若不添加该宏定义，则是根据课件中要求来记录“每次 cwnd 调整的时间和相应值”。具体而言，会在所有涉及 cwnd 改变的语句后加入`update_log`函数，记录当前时间（从套接字创建开始计算）和 cwnd 值，按行写入 log 文件中。具体的更新函数如下图所示。

```
void update_log(struct tcp_sock *tsk) {
#ifdef LOG_BY_TIME
    return;
#endif
    struct timeval current;
    gettimeofday(&current, tz: NULL);
    long duration = 1000000 * ( current.tv_sec - start.tv_sec ) + current.tv_usec - start.tv_usec;
    char line[100];
    sprintf(line, format: "%ld\t%d\n", duration, tsk->cwnd);
    fwrite(line, sizeof(char), strlen(line), record_file);
}
```

但是实际运行后发现，由于超时重传中 cwnd 并不马上改变，而是维持一段时间后突减为 1，所以得到的数据在 Excel 软件中拟合的结果并不很能反映超时重传情况下 cwnd 的实际变化趋势。如下图所示，每次 cwnd 减为 1 MSS 大小前的曲线都被拟合为从上一个值开始固定斜率的直线，并不符合实际行为。



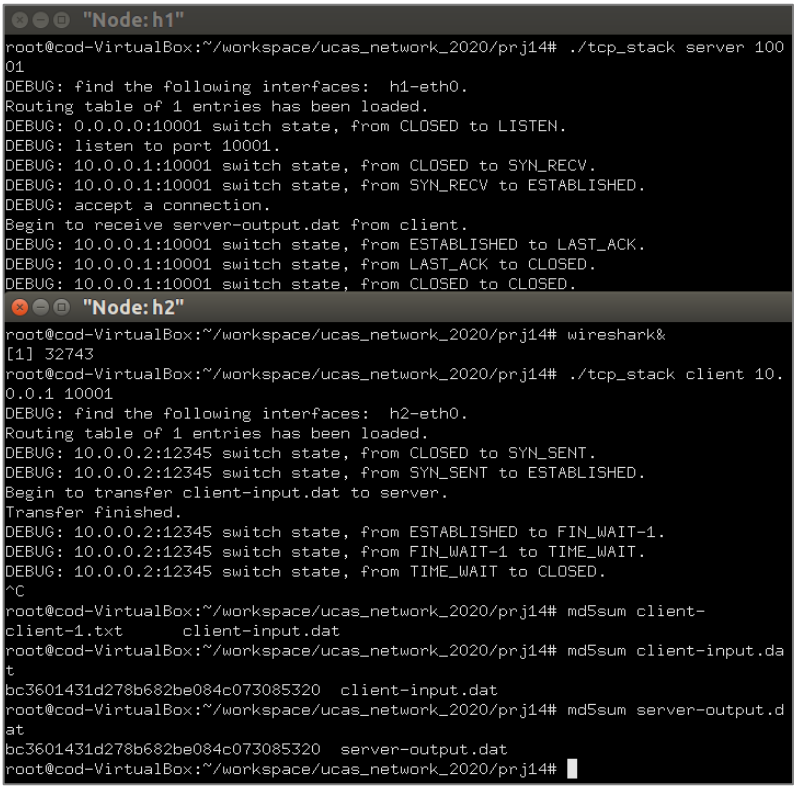
针对这一现象，本项目中还额外加入了定时记录的方法。即在 socket 创建时同时创建一个记录线程，每 1 ms 唤醒一次，记录此时的毫秒数和 cwnd 值。

```
void *tcp_record_thread(){
    int round = 1;
    while (1) {
        usleep( useconds: 1000);
        char line[100];
        sprintf(line, format: "%d\t%d\n", round++, eg->cwnd);
        fwrite(line, sizeof(char), strlen(line), record_file);
    }
}
```

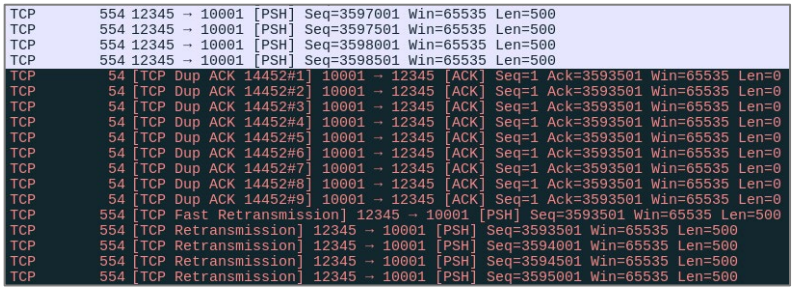
三、结果分析

（一）运行结果

如下图所示，执行` tcp_topo.py `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，可见传输正常，且 md5sum 的校验结果也正确。

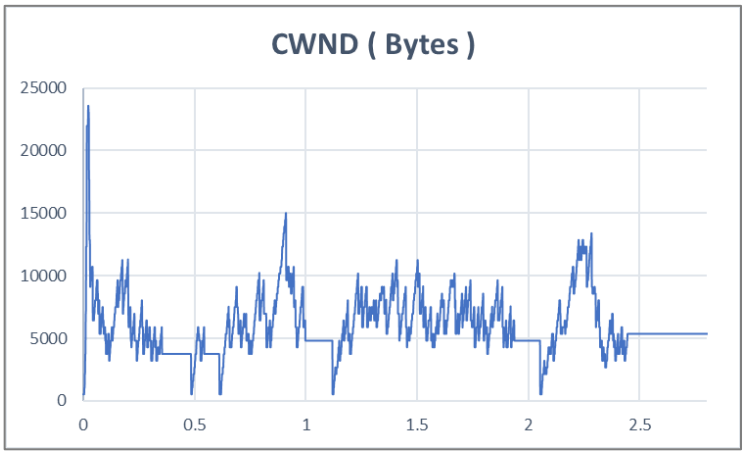


通过 wireshark 查看节点 h2 的收发包情况，可以看到数据包正常收发、快重传、超时重传等的过程和重复 ACK 的情况。



（二）cwnd 变化图及分析

根据实验中记录的数据（采用定时记录方法），通过 Excel 软件拟合得到 cwnd 基于时间变化的曲线图，如下所示。在作图时，对相应数据也进行了乘 MSS 的处理，所以纵坐标是以字节为单位的 cwnd 大小。对照快重传和超时重传的机制可以看出，拥塞窗口的变化可以说明拥塞控制机制基本正常运行。在快重传时 cwnd 值减半，超时重传时则降为最低值，超时重传出发前的超时等待时间内则不会发生变化。



虽然这种基于时间记录的方式可以简单地拟合出超时等待阶段 cwnd 不变的特征，但也不免会损失一些细节的变化。例如，由于数据包的确认往往是连续发生的（约 0.1 ms 的数量级），这种方式会忽略记录间隔中数据的变化规律，可能无法反映更细节的一些变化。如下图所示为在每次 cwnd 变化后记录的方法得到的部分数据，左侧为以秒为单位的时间，右侧为以字节为单位的 cwnd 大小。可以看出有许多变化会被基于时间间隔记录的方法给忽略。

0.011506	3752
0.018757	4288
0.018766	4824
0.018768	5360
0.01877	5896
0.018772	6432
0.018774	6968
0.022152	7504

（三）实验总结

本次是网络实验的最后一次实验。回望这一个学期，我们通过路由器、TCP 等实验，增进了对计算机网络理论课中很多知识点的理解，本次实验也不例外。

理论课中只是讲解了 cwnd 在不同机制下的变化方式，只是一条结果性质的知识。在理论课后，我也确实一直比较疑惑具体如何实现诸如“一个 RTT 后 cwnd 值加一 / 翻倍 / 减半”等操作：是通过浮点数存储吗？如果是的话在计算发送窗口时是取整还是将浮点部分也计算在内呢？减半时造成发送被阻塞的话该如何处理呢？类似的问题都在本次实验中得到了解决。

除此以外，在本次实验中，我也解决了一些之前实验中遗留的问题。例如对于无效 ACK 消息不应简单等同于重复 ACK 来处理、ring buffer 的溢出避免相关的锁机制。在上次实验的最后，我发现程序会时不时出现`coredump`的情况，但只要在启动时不发生，后续就可以正常运行。这次实验中我意识到原因在于多线程运行中部分变量申请空间的顺序不可预测，于是添加了一个作用类似于锁的全局变量，解决了该问题。

不过在实验调试中，有一个问题始终没有得到解决。在某些情况下，一端发送的 ACK 消息会凭空“消失”：在该端可以看到确实运行了 send 函数，但在同一端的 wireshark 无法监听到该包（不仅是对端无法监听到，所以可以排除链路丢包的可能）。猜测这是 mininet 底层某些机制的影响。