实验报告

网络传输机制实验四

一、实验内容

了解 TCP 的拥塞控制机制，了解 TCP 拥塞状态的转移过程，掌握数据包的发送条件、拥塞窗口的变化、快重传快恢复的实现方法，设计实现 TCP 的拥塞控制机制。

二、实验流程

1. 实现拥塞控制功能。
2. 在给定拓扑下验证拥塞控制的正确性。
3. 记录拥塞窗口值，画出拥塞窗口的变化曲线图。

三、实验结果及分析

（一）TCP 功能实现思路

1、修改一些已有函数功能

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)
```

函数，应用程序发送数据包。之前的实现是检测 snd_wnd 来判断是否能发送下一个数据包，现在改为根据在途数据包和发送窗口的大小来判断是否能发送下一个数据包。

```
int inflight = (tsk->snd_nxt - tsk->snd_una) / TCP_MSS - tsk->dupacks;
if (max(tsk->snd_wnd / TCP_MSS - inflight, 0) <= 0) {
    sleep_on(tsk->wait_send);
}
```

```
void tcp_update_window(struct tcp_sock *tsk, struct tcp_cb *cb)
```

函数，更新发送窗口。之前直接使用对方的接收窗口，现在改为接收窗口与拥塞窗口的最小值。

```
u16 old_snd_wnd = tsk->snd_wnd;
tsk->snd_wnd = min(cb->rwnd, tsk->cwnd * TCP_MSS);
if (old_snd_wnd <= 0)
    wake_up(tsk->wait_send);
```

```
void tcp_scan_retrans_timer_list(void)
```

函数，扫描重发定时器队列，对超时的定时器进行处理。在超时重发时，将拥塞状态变为 LOSS，将 ssthresh 减为拥塞窗口的一半，拥塞窗口减为 1，重新开始慢启动。

```
tsk->ssthresh = max(((u32)(tsk->cwnd / 2)), 1);
tsk->cwnd = 1;
tsk->nr_state = LOSS;
tsk->loss_point = tsk->snd_nxt;
time_entry->retrans_time += 1;
time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL * (1 << time_entry->retrans_time);
tcp_retrans_send_buffer(tsk);
```

2、拥塞状态转移

在 ESTABLISHED 状态下接收到 ACK 包时，使用 tcp_new_reno_process 函数处理拥塞状态变化和拥塞窗口改变。

```
void tcp_new_reno_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
```

函数，负责处理拥塞状态转移和拥塞窗口的变化。

```
void tcp_new_reno_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet) {
    int ack_valid = tcp_delete_send_buffer(tsk, cb->ack);

    switch (tsk->nr_state) {
        case OPEN: {
            if (tsk->cwnd < tsk->ssthresh)
                tsk->cwnd += 1;
            else
                tsk->cwnd += 1.0 / tsk->cwnd;
            if (!ack_valid) {
                tsk->dupacks ++;
                tsk->nr_state = DISORDER;
            }
            break;
        }
        case DISORDER: {
            if (tsk->cwnd < tsk->ssthresh)
                tsk->cwnd += 1;
            else
                tsk->cwnd += 1.0 / tsk->cwnd;
            if (!ack_valid) {
                tsk->dupacks ++;
                if (tsk->dupacks >= 3) {
                    tsk->ssthresh = max((u32)(tsk->cwnd / 2), 1);
                    tsk->cwnd -= 0.5;
                    tsk->cwnd_flag = 0;
                    tsk->recovery_point = tsk->snd_nxt;
                    tsk->nr_state = RECOVERY;
                    tcp_retrans_send_buffer(tsk);
                }
            }
            break;
        }
```

```
        case LOSS: {
            if (tsk->cwnd < tsk->ssthresh)
                tsk->cwnd += 1;
            else
                tsk->cwnd += 1.0 / tsk->cwnd;

            if (ack_valid) {
                if (cb->ack >= tsk->loss_point) {
                    tsk->nr_state = OPEN;
                    tsk->dupacks = 0;
                }
            }
            else {
                tsk->dupacks ++;
            }
            break;
        }
        case RECOVERY: {
            if (tsk->cwnd > tsk->ssthresh && tsk->cwnd_flag == 0)
                tsk->cwnd -= 0.5;
            else
                tsk->cwnd_flag = 1;

            if (ack_valid) {
                if (cb->ack < tsk->recovery_point) {
                    tcp_retrans_send_buffer(tsk);
                }
                else {
                    tsk->nr_state = OPEN;
                    tsk->dupacks = 0;
                }
            }
            else {
                tsk->dupacks ++;
                wake_up(tsk->wait_send);
            }
            break;
        }
        default:
            break;
    }

    tcp_update_retrans_timer(tsk);
}
```

首先使用 ack 对发送数据队列进行处理，将已经回复 ack 的数据包清理掉。如果有数据包被清理掉，说明该 ACK 是有效的，否则是重复的。

然后根据拥塞状态分别处理。

在 OPEN 状态，根据 cwnd 是否超过阈值来更新 cwnd。如果收到无效 ACK，切换到 DISORDER 状态。

在 DISORDER 状态，cwnd 更新与 OPEN 相同。在收到 3 个重复 ACK 后，判定丢包，启动快重传，切换到 RECOVERY 状态。

在 RECOVERY 状态，按照快恢复机制，cwnd 值变为新阈值，即原 cwnd 的一半，因此具体实现为每收到一个 ACK 下降 0.5cwnd。当所有的包都已确认接收，回到 OPEN 状态；否则重传一个包。如果 ACK 包是无效的，

dupacks 加一，在途数据包减少了一个，因此唤醒发送进程，可以再发送一个数据包。

LOSS 状态只能由超时重传触发，进入时阈值减为 cwnd 的一半，cwnd 减为 1，重新开始慢启动。LOSS 状态确认丢失的包全部接收后，转为 OPEN 状态。

最后处理完拥塞控制后，更新超时重发的计时器。

（二）实验功能验证

H1：本实验 server

H2：本实验 client

"Node: h1"

```
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 2920
DEBUG: write: 7080
DEBUG: write: 1460
DEBUG: write: 8540
DEBUG: write: 10000
DEBUG: write: 7300
DEBUG: write: 2700
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 1460
DEBUG: write: 8540
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 10000
DEBUG: write: 2632
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
```

"Node: h2"

```
DEBUG: retrans seq: 3920001

DEBUG: send: 10000, remain: 112632, total: (3940000/4052632)
DEBUG: send: 10000, remain: 102632, total: (3950000/4052632)
DEBUG: send: 10000, remain: 92632, total: (3960000/4052632)
DEBUG: send: 10000, remain: 82632, total: (3970000/4052632)
DEBUG: send: 10000, remain: 72632, total: (3980000/4052632)
DEBUG: retrans time: 1

DEBUG: retrans seq: 3977301

DEBUG: send: 10000, remain: 62632, total: (3990000/4052632)
DEBUG: send: 10000, remain: 52632, total: (4000000/4052632)
DEBUG: send: 10000, remain: 42632, total: (4010000/4052632)
DEBUG: send: 10000, remain: 32632, total: (4020000/4052632)
DEBUG: send: 10000, remain: 22632, total: (4030000/4052632)
DEBUG: send: 10000, remain: 12632, total: (4040000/4052632)
DEBUG: send: 10000, remain: 2632, total: (4050000/4052632)
DEBUG: send: 2632, remain: 0, total: (4052632/4052632)
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 FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

MD5 验证

```
joker@joker-linux:/mnt/hgfs/share/17-tcp_stack$ md5sum client-input.dat
d21b98a2c60b495f984e72b21d6794d1  client-input.dat
joker@joker-linux:/mnt/hgfs/share/17-tcp_stack$ md5sum server-output.dat
d21b98a2c60b495f984e72b21d6794d1  server-output.dat
```

综上可以看出，本实验拥塞控制功能正确，可以正常工作。

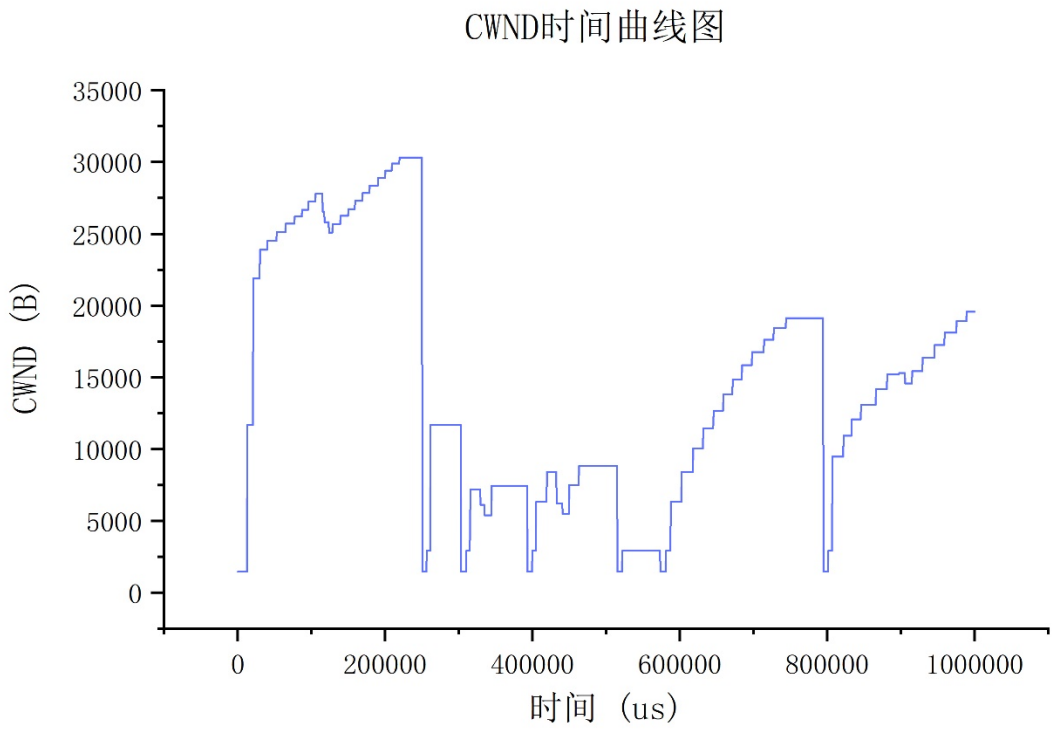
（三）拥塞窗口变化曲线图

使用一个单独的进程来监控 cwnd 拥塞窗口值的变化，该进程在 h2 client 进入 ESTABLISHED 状态后启动。最开始实现为仅在 cwnd 变化时记录数据，这样数据点较少，图像不好看。最后改为每 500 us 扫描一次 cwnd，记录数据。

```c
void *tcp_cwnd_thread(void *arg) {
    struct tcp_sock *tsk = (struct tcp_sock *)arg;
    FILE *fp = fopen("cwnd.txt", "w");

    int time_us = 0;
    while (tsk->state == TCP_ESTABLISHED && time_us < 1000000) {
        usleep(500);
        time_us += 500;
        fprintf(fp, "%d %f %f\n", time_us, tsk->cwnd, tsk->cwnd * TCP_MSS);
    }
    fclose(fp);
    return NULL;
}
```

作出 cwnd 时间曲线图如下：



可以看出，曲线走势基本与讲义上相同。其中 cwnd 稍微下降又迅速恢复的，比如 100000 us 处，是触发了快速重传与快恢复。而直接降至最低的，是触发了超时重传，又重新开始慢启动。

综上所述，本实验拥塞控制功能正确，可以正常工作。

四、实验总结

通过本次实验，我了解了 TCP 协议栈的拥塞控制功能，了解了如何根据在途数据包来控制数据包的发送，同时也掌握了实现快重传快恢复的方法，这让我对 TCP 协议有了更多的认识。