

# 网络传输机制实验三

学号： 2021K8009929010

姓名： 贾城昊

## 一、 实验题目：网络传输机制实验三

## 二、 实验任务：

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。

## 三、 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。

## 四、 实验过程

### （一）拥塞状态转移

#### 1. 拥塞控制算法总述

- 慢启动 (Slow Start)

- 对方每确认一个报文段, cwnd 增加 1MSS, 直到 cwnd 超过 ssthresh 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 MSS)/CWND \*1MSS
  
  - 经过 1 个 RTT，前一个 cwnd 的所有数据被确认后，cwnd 增加 1 MSS
  
- 快重传 (Fast Retransmission)
  - ssthresh 减小为当前 cwnd 的一半:  $ssthresh \leftarrow cwnd/2$
  
  - 新拥塞窗口值  $cwnd \leftarrow$  新的 ssthresh
  
- 超时重传 (Retransmission Timeout)
  - Ssthresh 减小为当前 cwnd 的一半:  $ssthresh \leftarrow cwnd/2$
  
  - 拥塞窗口值 cwnd 减为 1 MSS
  
- 快恢复 (Fast Recovery) : 以下简称 FR
  - 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

当触发超时重传后，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 infligh 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。

## 2. 拥塞状态机切换和拥塞窗口控制（具体介绍）

tcp\_congestion\_control 函数负责处理拥塞状态转移和拥塞窗口的变化。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。

首先需要判断收到的 ACK 是否有效。这个判断过程可以通过 ACK 遍历发送队列。在这个遍历的同时，可以对发送数据队列进行处理,将已经回复 ACK 的数据包清理掉。按照这个思路，只用修改以往实验实现的更新发送队列的函数，在更新的过程中，如果有数据包被清理掉,则说明该 ACK 是有效的。

```

//Update the TCP send buffer based on the acknowledgment number
int tcp_update_send_buffer(struct tcp_sock *tsk, u32 ack) {
    int ret = 0;

    send_buffer_entry_t *send_buffer_entry, *send_buffer_entry_q;
    pthread_mutex_lock(&tsk->send_buf_lock);

    list_for_each_entry_safe(send_buffer_entry, send_buffer_entry_q, &tsk->send_buf, list) {
        struct tcphdr *tcp = packet_to_tcp_hdr(send_buffer_entry->packet);
        u32 seq = ntohl(tcp->seq);

        // If the sequence number is less than the acknowledgment number, delete the entry
        if (less_than_32b(seq, ack)) {
            //log(DEBUG, "delete %d %d\n", seq, ack);
            list_delete_entry(&send_buffer_entry->list);
            free(send_buffer_entry->packet);
            free(send_buffer_entry);
            ret = 1;
        }
    }
    pthread_mutex_unlock(&tsk->send_buf_lock);

    return ret;
}

```

然后便可以根据拥塞状态分别处理收到的 ACK 包。如在 OPEN 状态,表面当网络中没有发生丢包,也就不需要重传,发送方按照慢启动或者拥塞避免算法处理到来的 ACK。即检查拥塞窗口是否小于慢启动阈值。如果是,通过一个固定步长增加拥塞窗口;否则,按照拥塞避免算法增加拥塞窗口。如果发现收到无效 ACK,则把拥塞状态切换到 DISORDER 状态,增加 dupacks 的值 (dupacks 的值代表接收方在接收到数据的时候发送的重复确认的数量)。

```

switch (tsk->c_state) {
    case OPEN: {
        if (tsk->cwnd < tsk->ssthresh)
            tsk->cwnd += 1;
        else
            tsk->cwnd += 1.0 / tsk->cwnd;
        if (!ack_valid) {
            tsk->dupacks++;
            tsk->c_state = DISORDER;
        }
        break;
    }
}

```

在 DISORDER 状态, cwnd 更新与 OPEN 相同。根据拥塞窗口是否小于慢启动阈

值来选择慢启动或者拥塞避免算法。如果接收到的 ACK 无效, 增加重复 ACK 计数 如果重复 ACK 次数达到 3 次, 则判定为丢包, 启动快重传, 立即重传可能丢失的数据分组, 而不需等待重传计时器超时。同时切换到 RECOVERY 状态, 启动快恢复。

```
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->c_state = RECOVERY;
            tcp_retrans_send_buffer(tsk);
        }
    }
    break;
}
```

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, cwnd 下降 0.5。当对方确认了进入 RECOVERY 前发送的所有数据时, 回到 OPEN 状态; 否则, 重传对应的数据包。

如果在 RECOVERY 状态收到的 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。

在下面代码中包括标志位 cwnd\_flag, 当 flag 为 0 时说明此时 cwnd 还需要继续下降。

```

case RECOVERY: {
    if (tsk->cwnd > tsk->sssthresh && 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->c_state = OPEN;
            tsk->dupacks = 0;
        }
    }
    else {
        tsk->dupacks++;
        wake_up(tsk->wait_send);
    }
    break;
}

```

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（在后文中会展示相关代码）。当接收方确认了进入 LOSS 前发送的所有数据时，转为 OPEN 状态。若接收到了无效的 ACK，则增加重复 ACK 计数。

```

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

    if (ack_valid) {
        if (cb->ack >= tsk->loss_point) {
            tsk->c_state = OPEN;
            tsk->dupacks = 0;
        }
    }
    else {
        tsk->dupacks++;
    }
    break;
}

```

最后处理完拥塞控制后，更新超时重发的计时器。

## (二) 已写函数的修改：实现拥塞控制

### 1. 发送窗口的维护

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拥塞窗口，而是直接使用接收窗口大小。本次实验中引入了拥塞窗口，发送窗口每次都应当更新为拥塞窗口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。

具体修改的代码如下所示：

```
// if the snd_wnd before updating is zero, notify tcp_sock_send (wait_send)
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(tsk->adv_wnd, tsk->cwnd * TCP_MSS);
    if (old_snd_wnd == 0)
        wake_up(tsk->wait_send);
}
```

### 2. 数据包发送函数的更新

tcp\_sock\_write 函数负责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，现在需要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。

具体添加的代码如下：

```
int inflight = (tsk->snd_nxt - tsk->snd_una) - tsk->dupacks * TCP_MSS;
if (max(tsk->snd_wnd - inflight, 0) <= 0) {
    log(DEBUG, "snd_wnd: %d; inflight: %d;", tsk->snd_wnd, inflight);
    sleep_on(tsk->wait_send);
}
```

### 3. 重发定时器队列扫描函数的更新

`tcp_scan_retrans_timer_list` 函数负责扫描重发定时器队列,对超时的定时器进行处理。为了实现拥塞控制,需要在超时重发时,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。

具体修改的代码如下:

```
else if (tsk->state != TCP_CLOSED) {
    time_entry->retrans_time += 1;
    log(DEBUG, "retrans time: %d\n", time_entry->retrans_time);

    tsk->ssthresh = max(((u32)(tsk->cwnd / 2)), 1);
    tsk->cwnd = 1;
    tsk->c_state = LOSS;
    tsk->loss_point = tsk->snd_nxt;
    update_log(tsk);

    time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL * (1 << time_entry->retrans_time);
    //time_entry->timeout = TCP_RETRANS_INTERVAL_INITIAL;
    tcp_retrans_send_buffer(tsk);
}
```

### 4. TCP 数据包与连接管理的更新

`tcp_process` 函数负责处理 TCP 数据包和连接管理。为了实现拥塞控制,在 `ESTABLISHED` 状态下接收到 `ACK` 包时,改为使用 `tcp_congestion_control` 函数处理拥塞状态变化和拥塞窗口改变(该函数功能在上面已经介绍过)。

具体修改的代码如下:



```

else{
    tsk->rcv_nxt = cb->seq_end;
    if (cb->ack > tsk->snd_una) {
        tsk->retrans_timer.retrans_time = 0;
        tsk->retrans_timer.timeout = TCP_RETRANS_INTERVAL_INITIAL;
    }
    //log(DEBUG, "%d %d\n", tsk->snd_una, cb->ack);
    tsk->snd_una = cb->ack;
    tcp_update_window_safe(tsk, cb);
    tcp_congestion_control(tsk, cb, packet);
    //tcp_update_send_buffer(tsk, cb->ack);
    //tcp_update_retrans_timer(tsk);
}

```

### (三) 统计 CWND 的变化

本项目中,提供了两种记录拥塞窗口变化情况的方法。可以通过 tcp.h 头文件中的宏定义 LOG\_AS\_PPT 来实现切换。若添加该宏定义,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 改变的语句后调用记录函数 cwnd\_record,记录当前时间(从发送方进入 ESTABLISH 状态开始计算)和 cwnd 值,按行写入 cwnd.txt 文件中。具体的更新函数如下图所示。

```

void cwnd_record(struct tcp_sock *tsk) {
#ifdef LOG_AS_PPT
    struct timeval current;
    gettimeofday(&current, NULL);
    long duration = 1000000 * ( current.tv_sec - start.tv_sec ) + current.tv_usec - start.tv_usec;
    char line[100];
    sprintf(line, "%d %f %f\n", duration, tsk->cwnd, tsk->cwnd * TCP_MSS);
    fwrite(line, sizeof(char), strlen(line), "cwnd.txt");
#else
    return;
#endif
}

```

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。每次 cwnd 减为 1 个 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。

为了解决这一问题,本实验中还额外加入了定时记录拥塞窗口大小的方法。即在发送方进入 ESTABLISH 状态同时创建一个记录线程,每 500us 唤醒一次,记录此时的微秒数和拥塞窗口的大小,具体代码如下图所示:

```
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;
}
```

## 五、实验结果与分析

### (一) 传输功能正确性验证

#### 1. 编写 Python 脚本

参考 tcp\_stack\_trans.py 编写 python 文件 tcp\_stack\_file.py, 使之能实现文件的传输, 具体的代码如下:

(注: 本次实验的 python 脚本与上次一样)

```

def server(port, filename):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    s.bind(('0.0.0.0', int(port)))
    s.listen(3)

    cs, addr = s.accept()
    print addr

    with open(filename, 'wb') as f:
        while True:
            data = cs.recv(1024)
            if data:
                f.write(data)
            else:
                break

    s.close()

def client(ip, port, filename):
    s = socket.socket()
    s.connect((ip, int(port)))

    f = open('client-input.dat', 'r')
    file_str = f.read()
    length = len(file_str)
    i = 0

    while length > 0:
        send_len = min(length, 100000)
        s.send(file_str[i: i+send_len])
        sleep(0.1)
        print("send:{0}, remain:{1}, total: {2}/{3}".format(i, length, i, i+length))
        length -= send_len
        i += send_len

    f.close()
    s.close()

if __name__ == '__main__':
    if sys.argv[1] == 'server':
        server(sys.argv[2], "server-output.dat")
    elif sys.argv[1] == 'client':
        client(sys.argv[2], sys.argv[3], "client-input.dat")

```

## 2. 本实验 Server 与本实验 Client

H1: 本实验 server

H2: 本实验 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab15/15-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:12345 from process to listen_queue
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: peer closed.
used time: 11 s
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:12345, state CLOSE_WAIT
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.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:12345.
```

```
"Node: h2"
DEBUG: sent 3904680 Bytes
DEBUG: retrans time: 1
DEBUG: retrans seq: 3890497
DEBUG: sent 3924704 Bytes
DEBUG: sent 3944728 Bytes
DEBUG: sent 3964752 Bytes
DEBUG: sent 3984776 Bytes
DEBUG: sent 4004800 Bytes
DEBUG: sent 4024824 Bytes
DEBUG: retrans seq: 3986237
DEBUG: sent 4044848 Bytes
DEBUG: sent 4052632 Bytes
DEBUG: the file has been sent completely.
DEBUG: close sock 10.0.0.2:12345 <-> 10.0.0.1:10001, state ESTABLISHED
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: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: Free 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

可以看到，虽然传输过程中出现了丢包的现象，但能正常进行重传，完成传输过程。并且传输时间花费了 11s，在预计时间范围内。

MD5 验证结果如下：

```
sai@Computer: ~/workspace/Network/lab15/15-tcp_stack
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb client-input.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum server-output.dat
05f7c5290651bce2fa0369f1921b3ccb server-output.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$
```

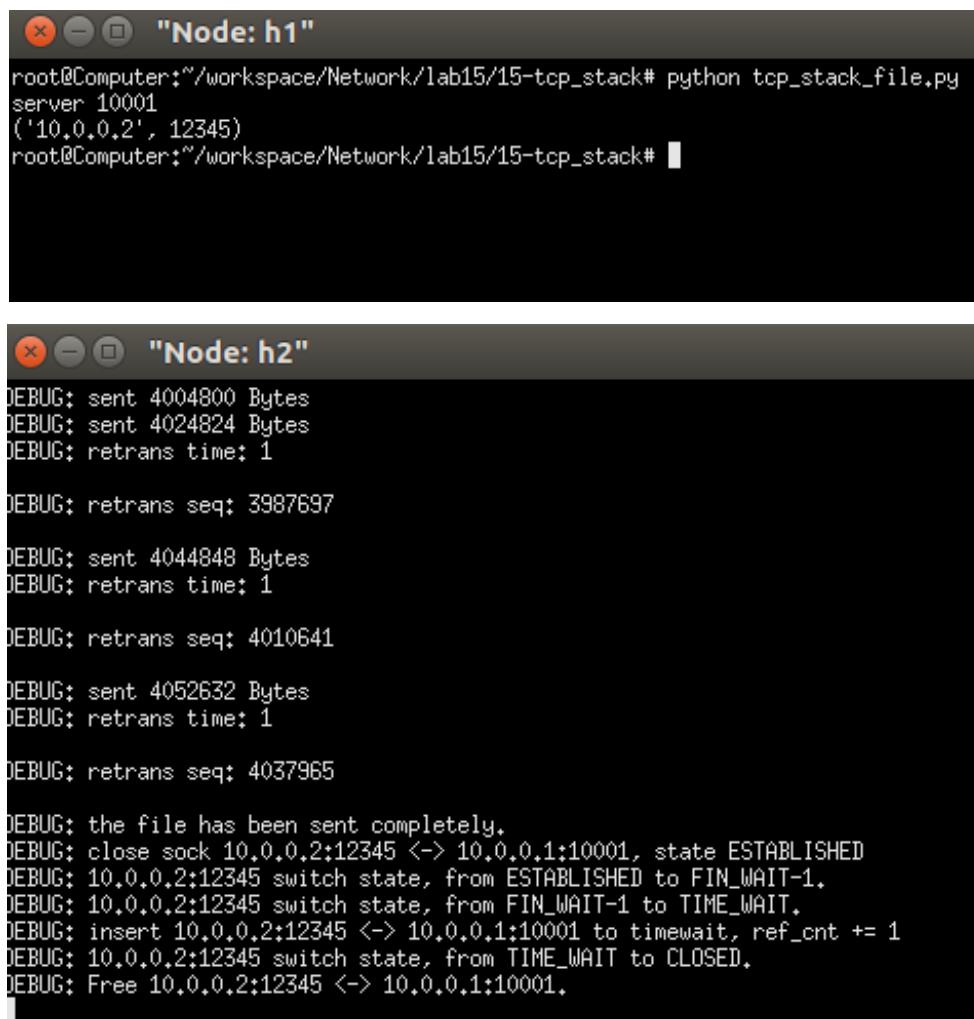
可以看出，最后服务器端接收到的文件与传输的文件一致。

### 3. 标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client

运行结果如下图所示:

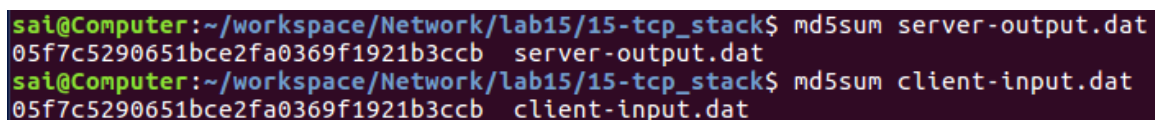


```
"Node: h1"
root@Computer:~/workspace/Network/lab15/15-tcp_stack# python tcp_stack_file.py
server 10001
('10.0.0.2', 12345)
root@Computer:~/workspace/Network/lab15/15-tcp_stack#

"Node: h2"
DEBUG: sent 4004800 Bytes
DEBUG: sent 4024824 Bytes
DEBUG: retrans time: 1
DEBUG: retrans seq: 3987697
DEBUG: sent 4044848 Bytes
DEBUG: retrans time: 1
DEBUG: retrans seq: 4010641
DEBUG: sent 4052632 Bytes
DEBUG: retrans time: 1
DEBUG: retrans seq: 4037965
DEBUG: the file has been sent completely.
DEBUG: close sock 10.0.0.2:12345 <-> 10.0.0.1:10001, state ESTABLISHED
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 TIME_WAIT.
DEBUG: insert 10.0.0.2:12345 <-> 10.0.0.1:10001 to timewait, ref_cnt += 1
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
DEBUG: Free 10.0.0.2:12345 <-> 10.0.0.1:10001.
```

可以看出, 虽然传输过程中出现了丢包的现象, 但能正常进行重传, 完成传输过程。

MD5 验证结果如下:



```
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum server-output.dat
05f7c5290651bce2fa0369f1921b3ccb server-output.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb client-input.dat
```

可以看出, 最后服务器端接收到的文件与传输的文件一致。

#### 4. 本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client

运行结果如下图所示:

```
"Node: h1"
root@Computer:~/workspace/Network/lab15/15-tcp_stack# ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: open file server-output.dat
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: listening port 10001.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: alloc a new tcp sock, ref_cnt = 1
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: Pass 10.0.0.1:10001 <-> 10.0.0.2:44076 from process to listen_queue
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: peer closed.
used time: 15 s
DEBUG: close this connection.
DEBUG: close sock 10.0.0.1:10001 <-> 10.0.0.2:44076, state CLOSE_WAIT
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.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to CLOSED.
DEBUG: Free 10.0.0.1:10001 <-> 10.0.0.2:44076.
```

```
"Node: h2"
send:1800000, remain:2252632, total: 1800000/4052632
send:1900000, remain:2152632, total: 1900000/4052632
send:2000000, remain:2052632, total: 2000000/4052632
send:2100000, remain:1952632, total: 2100000/4052632
send:2200000, remain:1852632, total: 2200000/4052632
send:2300000, remain:1752632, total: 2300000/4052632
send:2400000, remain:1652632, total: 2400000/4052632
send:2500000, remain:1552632, total: 2500000/4052632
send:2600000, remain:1452632, total: 2600000/4052632
send:2700000, remain:1352632, total: 2700000/4052632
send:2800000, remain:1252632, total: 2800000/4052632
send:2900000, remain:1152632, total: 2900000/4052632
send:3000000, remain:1052632, total: 3000000/4052632
send:3100000, remain:952632, total: 3100000/4052632
send:3200000, remain:852632, total: 3200000/4052632
send:3300000, remain:752632, total: 3300000/4052632
send:3400000, remain:652632, total: 3400000/4052632
send:3500000, remain:552632, total: 3500000/4052632
send:3600000, remain:452632, total: 3600000/4052632
send:3700000, remain:352632, total: 3700000/4052632
send:3800000, remain:252632, total: 3800000/4052632
send:3900000, remain:152632, total: 3900000/4052632
send:4000000, remain:52632, total: 4000000/4052632
root@Computer:~/workspace/Network/lab15/15-tcp_stack#
```

可以看出 H1 和 H2 能正常完成传输过程。并且传输只花费 15s, 也在预期时间范围内。

MD5 验证结果如下:

```

05f7c5290651bce2fa0369f1921b3ccb client-input.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb client-input.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$ md5sum client-input.dat
05f7c5290651bce2fa0369f1921b3ccb client-input.dat
sai@Computer:~/workspace/Network/lab15/15-tcp_stack$

```

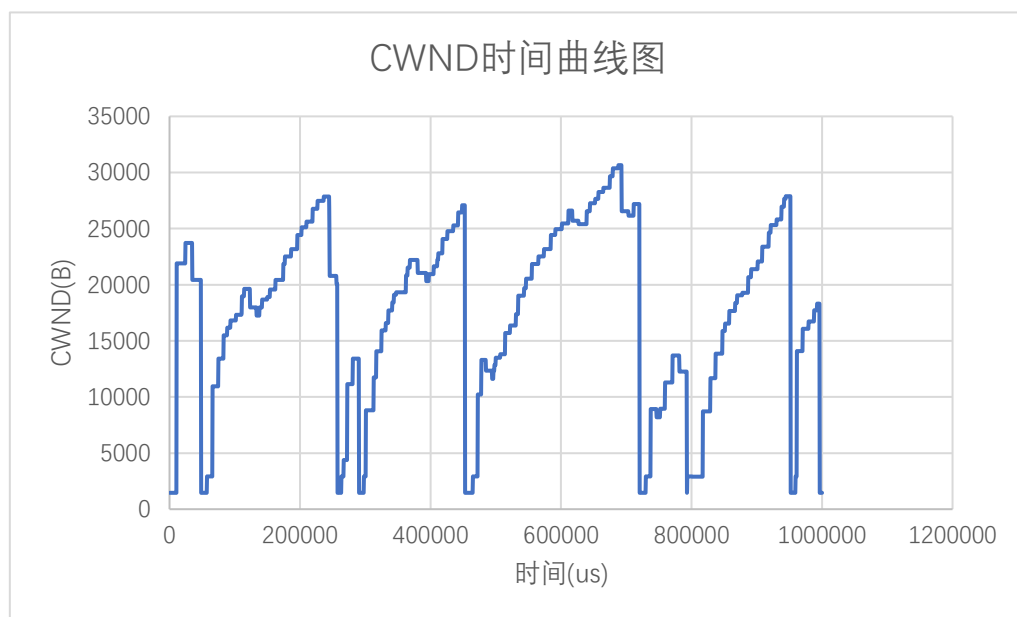
可以看出，最后服务器端接收到的文件与传输的文件一致。

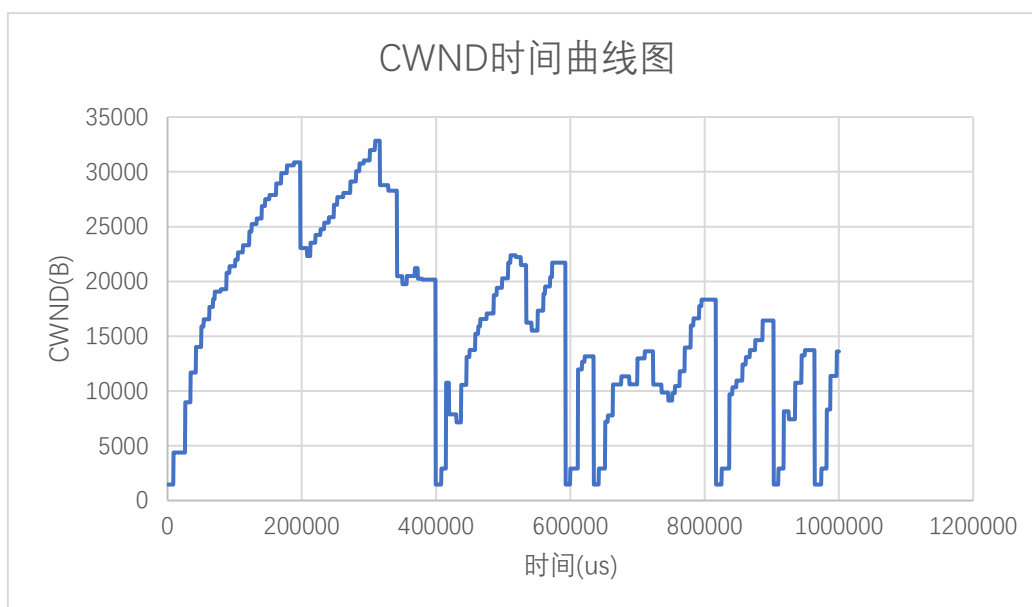
综上，本实验 server 和 client 能正确收发文件，功能正确。

## (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。先后进行了两次测量，得到了两次 cwnd 变化图，分别如下所示：

(注: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。其中 cwnd 下降又迅速恢复的,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。而超时重传出发前的超时等待时间内则不会发生变化。

## 六、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。

本次是网络传输机制实验的最后一次实验。回望这一个学期,我们实现了数据链路层的交换机,也实现了网络层的路由,最后完成 TCP 的一系列实验,在这个过程中,我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。但如何实现一个 RTT 后 cwnd 值加一/翻倍/减半的操作,减半时造成发送被阻塞的话该如何处理在理论课上均没有涉及。而这些类似的问题都在本次实验中都得到了解决

计算机网络实验即将结束,回顾整个学期,有面对实验框架代码理解的曲折,也有理论课知识逐渐清晰的感悟,有在 debug 时陷入绝望与无奈,也有最终成功通过 OJ 时的激动与兴奋。通过一整个学期的努力,完成了从数据链路层到传输层的协议,深刻体会了网络的魅力与精美,这段时间必定



会成为大学期间的难忘的一环。