网络传输机制实验四

中国科学院大学
袁欣怡 2018K8009929021
2021.7.8

实验内容

在之前的实验中，我们已经实现了2%丢包概率的网络环境下的文件传输。本次实验中，我们需要调整发送窗口`cwnd`的大小，来提高数据传输的速度，同时尽量避免网络拥塞。具体需要实现的内容包括：

1. 维护新增的变量：`TCP`拥塞控制状态和拥塞窗口大小`cwnd`。
2. 完成拥塞窗口增大的两个算法：慢启动和拥塞避免。
3. 完成拥塞窗口减小的两个算法：快重传和超时重传。
4. 完成拥塞窗口不变的一个算法：快恢复。

实验流程

1. 搭建实验环境

本实验中涉及到的文件主要有：

- `main.c`：编译后生成`tcp_stack`可执行文件，在两个`host`结点上运行。
- `Makefile`：处理终端`make all`和`make clean`命令。
- `tcp_in.c`：接收报文并处理需要的函数。本次实验修改其中`tcp_process`函数。

- `tcp_timer.c`：完成定时器相关的函数。本次实验中添加了定时保存当前拥塞窗口大小的函数`tcp_cwnd_plot_thread`。
- `tcp_topo_loss.py`：网络拓扑结构。
- `cwnd.txt`：通过`tcp_cwnd_plot_thread`函数保存的`cwnd`数据。
- `cwnd_result.py`：将`cwnd.txt`中的数据画成图像。

2. 实验代码设计

TCP拥塞控制状态

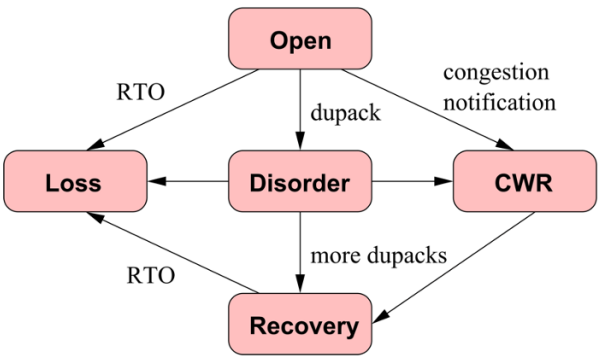
本次实验在数据结构`tcp_sock`中新增了拥塞控制状态`cstate`。`cstate`有四种状态：

- `Open`：网络中没有发生丢包，也没有收到重复`ACK`，证明网络状态良好。此状态下收到`ACK`后，适当增大`cwnd`。
- `Disorder`：网络中结点发现收到重复`ACK`。此状态下收到`ACK`后，适当增大`cwnd`。
- `Recovery`：网络结点发现丢包。此状态下需要重发丢失的包，且需要将`cwnd`减半。
- `Loss`：网络结点触发超时重传计时器，可以判断网络中发生严重丢包。此状态下认为所有未收到`ACK`的数据都丢失，重发这些丢失的包，并且让`cwnd`回到1，重新开始增长。
- `CWR`：网络结点收到`ECN`通知，将`cwnd`减半。此状态在本次实验中可以不使用，因而也未设计。

仿照`tcp_state`，在`tcp.h`添加拥塞控制状态：

```
// lab17 added:
// tcp congestion states
enum tcp_cstate {
    TCP_COPEN, TCP_CLOSS, TCP_CDISORDER, TCP_CFR, TCP_CRECOVERY
};
```

拥塞状态之间的转移规则如下图所示：



拥塞状态迁移的实现代码在“拥塞窗口大小变化”的末尾一并呈现。

拥塞窗口大小变化

改变`cwnd`的机制包括如下几种：

- 慢启动：在cwnd<ssthresh时，每收到一个`ACK`，cwnd++。且，经过1个`RTT`，前一个`cwnd`的所有数据被确认后， cwnd*=2。
- 拥塞避免：在cwnd>=ssthresh时，每收到一个`ACK`，cwnd+=1/cwnd。且，经过1个`RTT`，前一个`cwnd`的所有数据被确认后，cwnd++。
- 快重传：`ssthresh`和`cwnd`均减小为`cwnd/2`。
- 超时重传：`ssthresh`减小为`cwnd/2`，`cwnd`减小为1。
- 快恢复：快重传触发后立刻进行快恢复。此状态下，如果收到快恢复前发送的所有数据的`ACK`，则进入`Open`状态；如果触发超时重传，则进入`Loss`状态。快恢复期间如果收到`ACK`，再进行进一步分类讨论。

改变`cwnd`的代码主要在`tcp_in.c`中的`tcp_process`函数中实现。当`ESTABLISHED`状态下收到`ACK`时，进行控制状态迁移和`cwnd`的改变，因此此处只贴出这部分代码：

```
if (cb->pl_len == 0 || strcmp(cb->payload,"data_recv!") == 0){ // ACK
    tsk->snd_una = cb->ack;
    tsk->rcv_nxt = cb->seq +1;
    tcp_update_window_safe(tsk, cb);

    struct tcp_send_buffer_block *block;
    struct tcp_send_buffer_block *block_q;
    int delete = 0;

    // 遍历send_buffer，删除所有seq<ACK的数据缓存
    list_for_each_entry_safe(block, block_q, &send_buffer.list, list){
        struct iphdr *ip = packet_to_ip_hdr(block->packet);
        struct tcphdr *tcp = (struct tcphdr *)((char *)ip + IP_BASE_HDR_SIZE);
        int ip_tot_len = block->len - ETHER_HDR_SIZE;
        int tcp_data_len = ip_tot_len - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE;
        u32 seq = ntohl(tcp->seq);

        if (less_than_32b(seq, cb->ack)){
            pthread_mutex_lock(&send_buffer.lock);
            send_buffer.size -= tcp_data_len;
            list_delete_entry(&block->list);
            pthread_mutex_unlock(&send_buffer.lock);
            free(block->packet);
            free(block);
            delete = 1;
```

```
        }
    }

    if (delete == 1){ // 如果有缓存被删除，说明收到了新的ACK
        fprintf(stdout, "new ack arrive.\n");

        switch(tsk->cstate){
            case TCP_COPEN:
            case TCP_CDISORDER:
            case TCP_CRECOVERY:
            case TCP_CLOSS:
                if ((int)tsk->cwnd < tsk->ssthresh){ // Slow Start
                    ++tsk->cwnd;
                    fprintf(stdout,"slow start: cwnd + 1\n");
                    fprintf(stdout,"cwnd: %f \n",tsk->cwnd);
                }
                else{ // Congestion Avoidance
                    tsk->cwnd += 1/tsk->cwnd;
                    fprintf(stdout,"congestion avoidance: cwnd + 1/cwnd\n");
                    fprintf(stdout,"cwnd: %f \n",tsk->cwnd);
                }
                if (tsk->cstate!=TCP_CLOSS || cb->ack >= tsk->losspoint)
                    // ack>=losspoint说明超时重传时发送的报文均已收到ACK，可以进入Open状态
                    tsk->cstate = TCP_COPEN;
                break;
            case TCP_CFR:
                if (tsk->fr_flag == 1) // cwnd增大
                    tsk->cwnd += 1/tsk->cwnd;
                else if (tsk->cwnd > tsk->ssthresh) // cwnd减小
                    tsk->cwnd -= 0.5;
                else tsk->fr_flag = 1; // cwnd<=ssthresh，可以停止减小，置fr_flag为1

                if (cb->ack < tsk->recovery_point)
                    // 说明快重传时发送的报文尚未全部收到ACK
                    send_buffer_RETRAN_HEAD(tsk);
                else{
                    fprintf(stdout,"Fast Recovery over.\n");
                    tsk->cstate = TCP_COPEN;
                }
                break;
            default:
                break;
```

```
        }
    }
    else{ // 没有缓存被删除
        switch(tsk->cstate){
            case TCP_COPEN:
            case TCP_CDISORDER:
            case TCP_CLOSS:
                if ((int)tsk->cwnd < tsk->ssthresh){ // Slow Start
                    ++tsk->cwnd;
                    fprintf(stdout,"slow start: cwnd + 1\n");
                    fprintf(stdout,"cwnd: %f \n",tsk->cwnd);
                }
                else{ // Congestion Avoidance
                    tsk->cwnd += (1/tsk->cwnd);
                    fprintf(stdout,"congestion avoidance: cwnd +
1/cwnd\n");
                    fprintf(stdout,"cwnd: %f \n",tsk->cwnd);
                }
                if(tsk->cstate == TCP_COPEN)
                    tsk->cstate = TCP_CDISORDER;
                else if(tsk->cstate == TCP_CDISORDER)
                    tsk->cstate = TCP_CRECOVERY;
                break;
            case TCP_CRECOVERY:
                fprintf(stdout,"Fast Recovery active.\n");
                tsk->ssthresh = max(((u32)(tsk->cwnd / 2)), 1);
                tsk->cwnd -= 0.5;
                tsk->fr_flag = 0;
                tsk->recovery_point = tsk->snd_nxt;
                send_buffer_RETRAN_HEAD(tsk);
                tsk->cstate = TCP_CFR;
            case TCP_CFR:
                if (tsk->fr_flag == 1)
                    tsk->cwnd += 1/tsk->cwnd;
                else if (tsk->cwnd > tsk->ssthresh)
                    tsk->cwnd -= 0.5;
                else tsk->fr_flag = 1;
                break;
            default:
                break;
        }
    }
    tcp_update_retrans_timer(tsk);
    wake_up(tsk->wait_send);
    return;
}
```

```
else{ // data
    tcp_recv_data(tsk, cb, packet);
    return ;
}
```

保存拥塞窗口大小并画图

在`tcp_timer.c`中增加函数`tcp_cwnd_plot_thread`，用来新建一个保存`cwnd`的线程：

```
// lab17 added:
// write cwnd into cwnd.txt to make a plot
void *tcp_cwnd_plot_thread(void *arg)
{
    struct tcp_sock *tsk = (struct tcp_sock *)arg;
    FILE *file = fopen("cwnd.txt", "w");
    float i = 0;

    while (tsk->state != TCP_TIME_WAIT) {
        usleep(5);
        ++i;
        fprintf(file, "%f:%f\n",i/10000, tsk->cwnd);
    }

    fclose(file);
    return NULL;
}
```

用`cwnd_result.py`进行图像绘制：

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

def readfile(filename):
    data_list = []
    data_num = 0
    with open(filename, 'r') as f:
        for line in f.readlines():
            linestr = line.strip('\n')
            data_tuple = linestr.split(':')
            data_list.append(data_tuple)
            data_num += 1

    return data_list, data_num

data_list,num = readfile("./cwnd.txt")
x_list = [t[0] for t in data_list]
```

```
y_list = [t[1] for t in data_list]

x_list = list(map(float, x_list))
y_list = list(map(float, y_list))

plt.plot(x_list, y_list)
# plt.xlim(0,3)
plt.show()
```

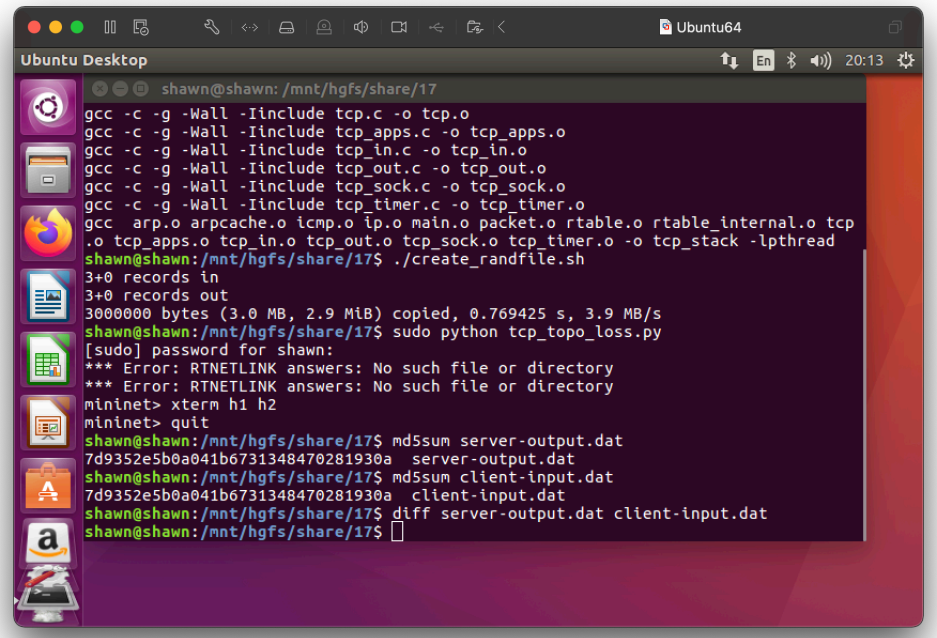
3. 启动脚本进行测试

传输过程：

> make clean
>
> make all
>
> ./create_randfile.sh
>
> sudo python tcp_topo_loss.py
>
> mininet> xterm h1 h2
>
> `h1#` ./tcp_stack server 10001
>
> `h2#` ./tcp_stack client 10.0.0.1 10001

检验结果：

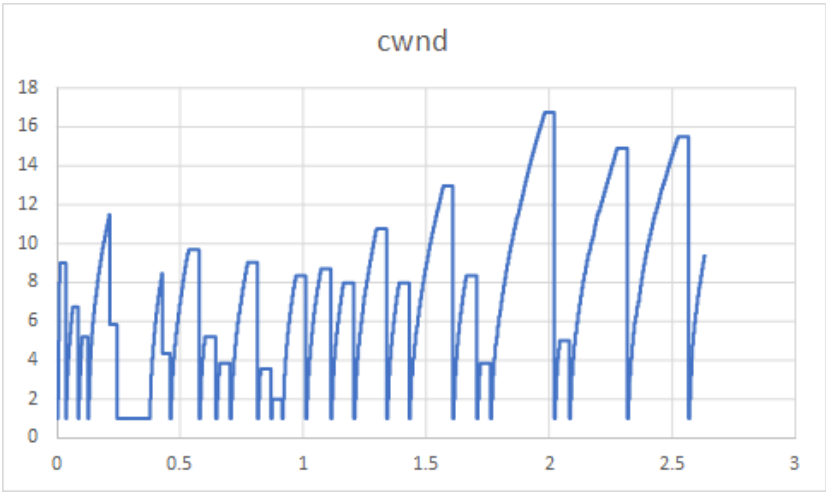
> md5sum server_output.dat
>
> md5sum client_input.dat
>
> diff server_output.dat client_input.dat



成功传输。

绘制图像：

```
python cwnd_result.py
```

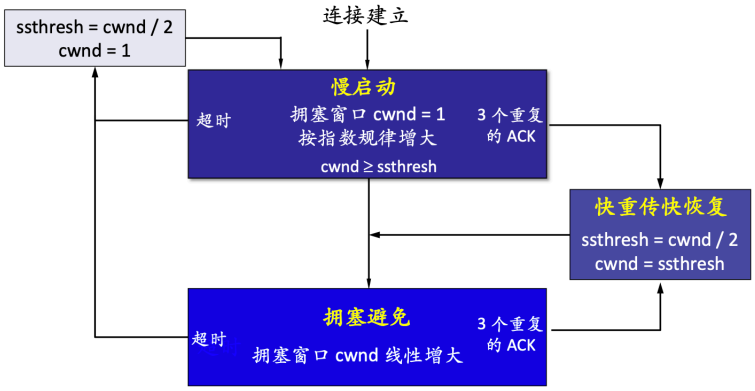


和理论基本一致。

实验总结与思考题

1. 一个疑问：拥塞避免时，`cwnd`的增量为`1/cwnd`，因此我将`cwnd`的类型修改为`float`，不理解为何实验框架中设置为`u32`？
2. 感觉将拥塞控制状态按照理论课ppt来设置更简单些，如下图：

TCP拥塞控制算法



3. 经过本次实验增加的拥塞控制后，丢包重传的次数与之前相比有下降，说明拥塞控制功能能很好地提升网络的传输效率，降低重传次数。