网络传输机制实验二报告

热伊莱·图尔贡 2018K8009929030

一、实验题目：网络传输机制实验二

二、实验内容

内容一：丢包恢复

·实现基于超时重传的TCP可靠数据传输，使得节点之间在有丢包网络中能够建立连接并正确传输数据

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)
- 在节点h1上执行TCP程序
    - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)
- 在节点h2上执行TCP程序
    - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)，禁止协议栈的相应功能
    - 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)
        - Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat
- 使用md5sum比较两个文件是否完全相同
- 使用tcp_stack.py替换两端任意一方，对端都能正确处理数据收发

内容二：拥塞控制

·实现TCP NewReno拥塞控制机制，发送方能够根据网络拥塞（丢包）信号调整拥塞窗口大小

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)
- 在节点h1上执行TCP程序
  - 执行脚本(disable_offloading.sh , disable_tcp_rst.sh)，禁止协议栈的相应功能
  - 在h1上运行TCP协议栈的服务器模式 (./tcp_stack server 10001)
- 在节点h2上执行TCP程序
  - 执行脚本(disable_offloading.sh, disable_tcp_rst.sh)，禁止协议栈的相应功能
  - 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)
    - Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat
- 使用md5sum比较两个文件是否完全相同
- 记录h2中每次cwnd调整的时间和相应值，呈现到二维坐标图中

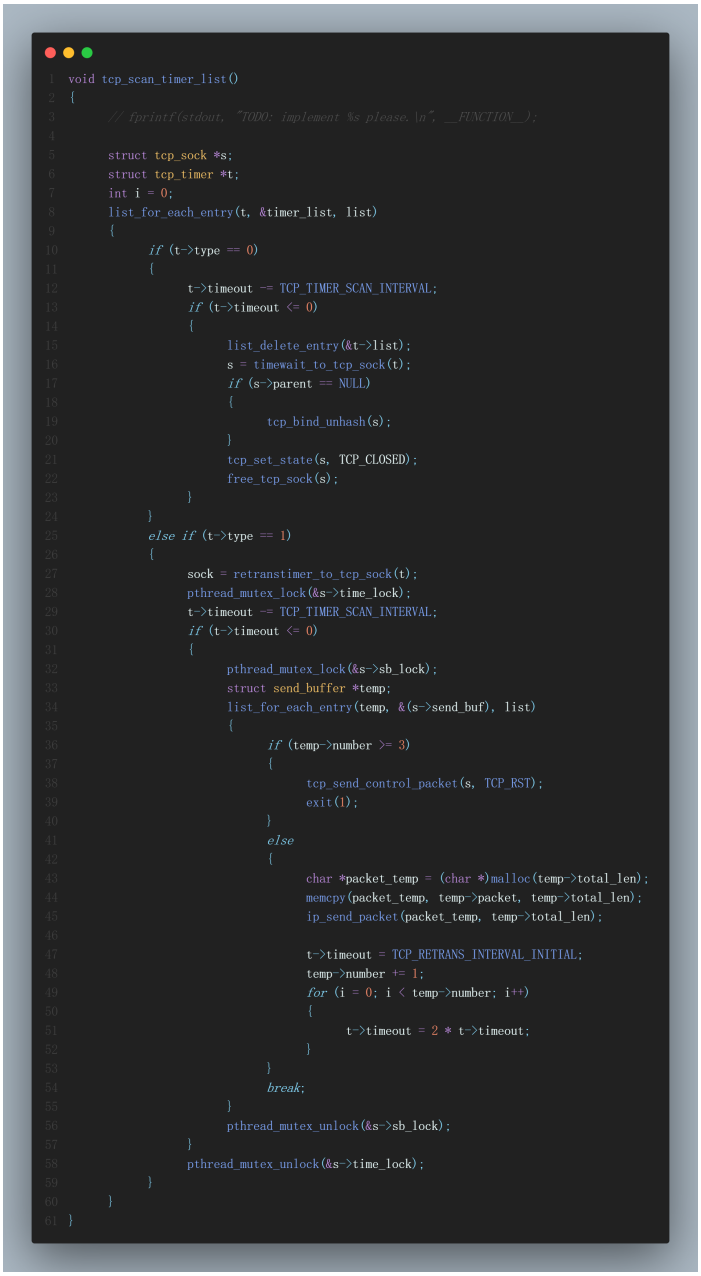
三、实验过程

- 本次实验中，需要通过实现超时重传机制以及发送和接收队列的维护来支持TCP可靠数据传输

超时重传机制

- 每个连接维护一个超时重传定时器
- 定时器管理
  - 当发送一个带数据/SYN/FIN的包，如果定时器是关闭的，则开启并设置时间为200ms
  - 当ACK确认了部分数据，重启定时器，设置时间为200ms
  - 当ACK确认了所有数据/SYN/FIN，关闭定时器
- 触发定时器后
  - 重传第一个没有被对方连续确认的数据/SYN/FIN
  - 定时器时间翻倍，记录该数据包的重传次数
  - 当一个数据包重传3次，对方都没有确认，关闭该连接(RST)
- 超时重传实现
  - 在tcp_sock中维护定时器

        - struct tcp_timer retrans_timer;
    - 当开启定时器时
        - 将retrans_timer放到timer_list中
    - 关闭定时器时
        - 将retrans_timer从timer_list中移除
    - 定时器扫描
        - 建议每10ms扫描一次定时器队列，重传定时器的值为200ms * 2^N



```
void tcp_scan_timer_list()
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

    struct tcp_sock *s;
    struct tcp_timer *t;
    int i = 0;
    list_for_each_entry(t, &timer_list, list)
    {
        if (t->type == 0)
        {
            t->timeout -= TCP_TIMER_SCAN_INTERVAL;
            if (t->timeout <= 0)
            {
                list_delete_entry(&t->list);
                s = timewait_to_tcp_sock(t);
                if (s->parent == NULL)
                {
                    tcp_bind_unhash(s);
                }
                tcp_set_state(s, TCP_CLOSED);
                free_tcp_sock(s);
            }
        }
        else if (t->type == 1)
        {
            sock = retranstimer_to_tcp_sock(t);
            pthread_mutex_lock(&s->time_lock);
            t->timeout -= TCP_TIMER_SCAN_INTERVAL;
            if (t->timeout <= 0)
            {
                pthread_mutex_lock(&s->sb_lock);
                struct send_buffer *temp;
                list_for_each_entry(temp, &(s->send_buf), list)
                {
                    if (temp->number >= 3)
                    {
                        tcp_send_control_packet(s, TCP_RST);
                        exit(1);
                    }
                    else
                    {
                        char *packet_temp = (char *)malloc(temp->total_len);
                        memcpy(packet_temp, temp->packet, temp->total_len);
                        ip_send_packet(packet_temp, temp->total_len);

                        t->timeout = TCP_RETRANS_INTERVAL_INITIAL;
                        temp->number += 1;
                        for (i = 0; i < temp->number; i++)
                        {
                            t->timeout = 2 * t->timeout;
                        }
                    }
                    break;
                }
                pthread_mutex_unlock(&s->sb_lock);
            }
            pthread_mutex_unlock(&s->time_lock);
        }
    }
}
```

发送队列

- 所有未确认的数据/SYN/FIN包，在收到其对应的ACK之前，都要放在发送队列snd_buffer（链表实现）中，以备后面可能的重传
- 发送新的数据时
    - 放到snd_buffer队尾，打开定时器
- 收到新的ACK

    - 将snd_buffer中seq_end <= ack的数据包移除，并更新定时器
- 重传定时器触发时
    - 重传snd_buffer中第一个数据包，定时器数值翻倍

```
struct send_buffer
{
        struct list_head list;
        char *packet;
        u32 seq;
        int data_len;
};
```

接收队列

- 数据接收方需要维护两个队列
    - 已经连续收到的数据，放在rcv_ring_buffer中供app读取
    - 收到不连续的数据，放到rcv_ofo_buffer队列（链表实现）中
- TCP属于发送方驱动传输机制
    - 接收方只负责在收到数据包时回复相应ACK
- 收到不连续的数据包时
    - 放在rcv_ofo_buffer队列，如果队列中包含了连续数据，则将其移到rcv_ring_buffer中

```c
void handle_rcv_data(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
    if (cb->seq == tsk->rcv_nxt)
    {
        pthread_mutex_lock(&(tsk->rcv_buf->rb_lock));
        write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
        pthread_mutex_unlock(&(tsk->rcv_buf->rb_lock));

        tsk->rcv_nxt += cb->pl_len;
        u32 record = tsk->rcv_nxt;
        while (1)
        {
            int flag = 0;
            struct recv_buffer *temp;
            list_for_each_entry(temp, &tsk->rcv_ofo_buf, list)
            {
                if (temp->seq == record)
                {
                    struct tcphdr *tcphdr_t = packet_to_tcp_hdr(temp->packet);
                    char *data_t = (char *)tcphdr_t + tcphdr_t->off * 4;
                    pthread_mutex_lock(&(tsk->rcv_buf->rb_lock));
                    write_ring_buffer(tsk->rcv_buf, data_t, temp->datalen);
                    pthread_mutex_unlock(&(tsk->rcv_buf->rb_lock));
                    tsk->rcv_nxt += temp->datalen;
                    record = tsk->rcv_nxt;
                    flag = 1;
                    list_delete_entry(&temp->list);
                    break;
                }
            }
            if (flag == 0)
            {
                break;
            }
        }
        tcp_send_control_packet(tsk, TCP_ACK);
        if (tsk->wait_recv->sleep == 1)
        {
            wake_up(tsk->wait_recv);
        }
    }
    else if (less_than_32b(tsk->rcv_nxt, cb->seq))
    {
        struct iphdr *iphdr_t = packet_to_ip_hdr(packet);
        int allen = ntohs(iphdr_t->tot_len) + ETHER_HDR_SIZE;
        struct recv_buffer *temp = (struct recv_buffer *)malloc(sizeof(struct recv_buffer));

        char *packet_temp = (char *)malloc(allen);
        memcpy(packet_temp, packet, allen);

        temp->packet = packet_temp;
        temp->total_len = allen;
        temp->seq = cb->seq;
        temp->datalen = cb->pl_len;

        list_add_tail(&temp->list, &tsk->rcv_ofo_buf);
    }
    else
    {
        tcp_send_control_packet(tsk, TCP_ACK);
    }
    tsk->snd_una = cb->ack - 1;
    delete_send_buf(tsk, cb->ack);
}
```

拥塞控制

·TCP拥塞窗口增大

- 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

```c
void update_cwnd(struct tcp_sock *tsk)
{
  if ((int)tsk->cwnd < tsk->ssthresh)
  {
    tsk->cwnd ++;
  }
```

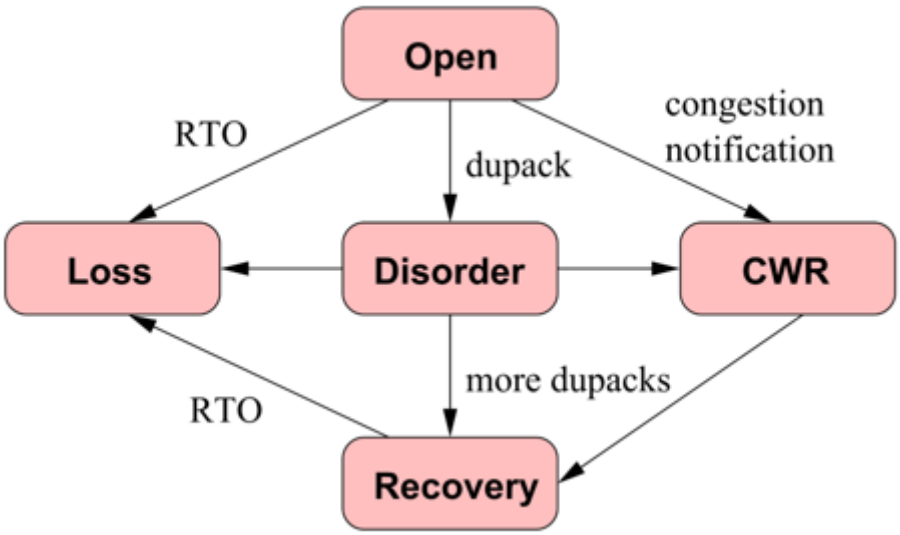
```
    else
    {
      tsk->cwnd += 1.0/tsk->cwnd;
    }
  }
```

·TCP拥塞窗口减小

- 快重传（Fast Retransmission）
  - Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2
  - 新拥塞窗口值cwnd <- 新的ssthresh
- 超时重传（Retransmission Timeout）
  - Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2
  - 拥塞窗口值cwnd减为1 MSS

·TCP拥塞窗口不变

- 快恢复（Fast Recovery）
  - 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RTO后，进入Loss状态
  - 在FR内，收到一个ACK：
    - 如果该ACK没有确认新数据，则说明inflight减一，cwnd允许发送一个新数据包
    - 如果该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



·TCP拥塞控制状态迁移

- Open: 没有丢包/重复ACK

    - 收到ACK后增加拥塞窗口值

```
if (tsk->current_state == TCP_OPEN)
{
  if (isNewAck)
  {
    update_cwnd(tsk);
  }
  else
  {
    tsk->dupacks ++;
    update_cwnd(tsk);
    tsk->current_state = TCP_DISORDER;
  }
  return;
}
```

- Disorder: 收到重复ACK，不够触发重传
    - 同Open状态
- CWR: 收到ECN通知
    - 窗口大小减半
- Recovery: 遇到网络丢包
    - 窗口值减半，恢复丢包
- Loss: 触发超时重传定时器
    - 认为所有未确认的数据都丢失
    - 窗口从1开始慢启动增长

四、实验结果

实验结果出错。