- 网络传输机制实验（三）
  - 实验内容
  - 实验步骤
  - 设计思路
    - 超时重传实现
      - `tcp_set_retrans_timer`、`tcp_update_retrans_timer`、tcp_unset_retrans_timer 函数
      - `tcp_scan_retrans_timer_list` 函数
      - `retrans_send_buffer_packet` 函数
    - 发送队列维护
      - `add_send_buffer_entry`、`alloc_send_buffer_entry` 函数
      - `delete_send_buffer_entry` 函数
    - 接收队列维护
      - `add_recv_ofo_buf_entry` 函数
      - `put_recv_ofo_buf_entry_to_ring_buf` 函数
  - 结果验证
  - 遇到的问题

网络传输机制实验（三）

2021年1月6日

蔡润泽

本实验 `Github` 地址

实验内容

支持TCP可靠数据传输

- 网络丢包
- 超时重传机制
- 有丢包场景下的连接建立和断开
- 发送队列和接收队列
- 超时定时器实现

实验步骤

- 修改tcp_apps.c(以及tcp_stack.py)，使之能够收发文件
- 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.py)
- 在节点h1上执行TCP程序
  - 执行脚本(disable_tcp_rst.sh, disable_offloading.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_tcp_rst.sh, disable_offloading.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替换其中任意一端，对端都能正确收发数据

设计思路

- 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，每10ms扫描一次定时器队列，重传定时器的值为200ms * 2^N。

`tcp_set_retrans_timer`、`tcp_update_retrans_timer`、tcp_unset_retrans_timer 函数

分别负责定时器的设置、更新以及删除。

`tcp_scan_retrans_timer_list` 函数

该函数负责扫描符合条件的定时器，若超时，则判断是否需要重传。

若没有超过重传次数上界（本设计中设计为5），则调用`retrans_send_buffer_packet`函数，重传发送队列snd_buffer中第一个数据包。

`retrans_send_buffer_packet` 函数

负责重传send buffer中第一个数据包。

发送队列维护

- 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。

上述两步在本设计中通过修改tcp_out.c中的相关函数实现。

- 收到新的ACK，将snd_buffer中已经确认的数据包移除，并更新定时器。 该步骤通过在 `tcp_process` 函数修改相关状态下的处理流程，从而被调用。
- 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。

`add_send_buffer_entry` 、 `alloc_send_buffer_entry` 函数

`alloc_send_buffer_entry` 负责创建一个buffer项，并在 `add_send_buffer_entry` 添加到snd_buffer队尾。

具体实现如下：

```
tcp_send_buffer_entry_t * alloc_send_buffer_entry(char *packet, int len) {
        tcp_send_buffer_entry_t * entry = (tcp_send_buffer_entry_t *)malloc(sizeof(tcp_
        bzero(entry, sizeof(tcp_send_buffer_entry_t));
        entry->packet = (char *)malloc(len);
        memcpy((char*)entry->packet, packet, len);
        entry->len = len;
        return entry;
}

void add_send_buffer_entry(struct tcp_sock *tsk, char *packet, int len) {
        tcp_send_buffer_entry_t * entry = alloc_send_buffer_entry(packet, len);
        list_add_tail(&entry->list, &tsk->send_buf);
}
```

其中， `tcp_send_buffer_entry_t` 的数据结构如下：

```
typedef struct {
        struct list_head list;
        char * packet;
        int len;
} tcp_send_buffer_entry_t;
```

`delete_send_buffer_entry` 函数

该函数负责移除已确认的发送数据包，具体实现如下：

```
void delete_send_buffer_entry(struct tcp_sock *tsk, u32 ack) {
        //printf("Delete a send_buffer_entry here.\n");
        tcp_send_buffer_entry_t * entry, * entry_q;
        list_for_each_entry_safe(entry, entry_q, &tsk->send_buf, list) {
                struct tcphdr *tcp = packet_to_tcp_hdr(entry->packet);
                u32 seq = ntohl(tcp->seq);
                if (less_than_32b(seq, ack)) {
                        list_delete_entry(&entry->list);
                        free(entry->packet);
                        free(entry);
                }
        }
}
```

接收队列维护

- 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。

`add_recv_ofo_buf_entry` 函数

该函数负责将收到的不连续数据放到rcv_ofo_buffer队列中，具体实现如下：

```
void add_recv_ofo_buf_entry(struct tcp_sock *tsk, struct tcp_cb *cb) {
        rcv_ofo_buf_entry_t * latest_ofo_entry = (rcv_ofo_buf_entry_t *)malloc(sizeof(r
        latest_ofo_entry->seq = cb->seq;
        latest_ofo_entry->len = cb->pl_len;
        latest_ofo_entry->data = (char*)malloc(cb->pl_len);
        memcpy(latest_ofo_entry->data, cb->payload, cb->pl_len);
        rcv_ofo_buf_entry_t * entry, *entry_q;
        list_for_each_entry_safe (entry, entry_q, &tsk->rcv_ofo_buf, list) {
                if(less_than_32b(latest_ofo_entry->seq , entry->seq)) {
                        list_add_tail(&latest_ofo_entry->list, &entry->list);
                        return;
                }
        }
        list_add_tail(&latest_ofo_entry->list, &tsk->rcv_ofo_buf);
}
```

其中，`rcv_ofo_buf_entry_t` 的数据结构如下：

```
typedef struct {
        struct list_head list;
        char * data;
        int len;
        int seq;
} rcv_ofo_buf_entry_t;
```

`put_recv_ofo_buf_entry_to_ring_buf` 函数

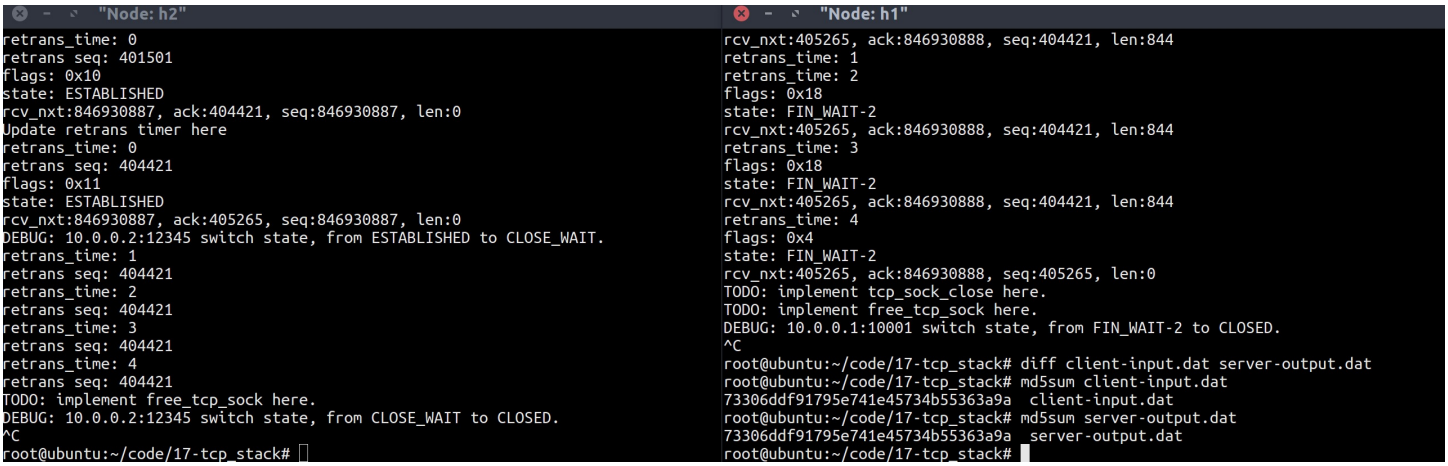
该函数负责将已经连续收到的数据，放在rcv_ring_buffer中供app读取，具体实现如下：

```
int put_recv_ofo_buf_entry_to_ring_buf(struct tcp_sock *tsk) {
        u32 seq = tsk->rcv_nxt;
        rcv_ofo_buf_entry_t * entry, * entry_q;
        list_for_each_entry_safe(entry, entry_q, &tsk->rcv_ofo_buf, list) {
                if (seq == entry->seq) {
                        while(entry->len > ring_buffer_free(tsk->rcv_buf)) {
                                sleep_on(tsk->wait_recv);
                        }
                        write_ring_buffer(tsk->rcv_buf, entry->data, entry->len);
                        wake_up(tsk->wait_recv);
                        seq += entry->len;
                        tsk->rcv_nxt = seq;
                        list_delete_entry(&entry->list);
                        free(entry->data);
                        free(entry);
                } else if (less_than_32b(seq, entry->seq)) {
                        break;
                } else {
                        return -1;
                }
        }
        return 0;
}
```

结果验证

由于实现简单重传机制非常耗时，此实验将传输文本大小缩短到了500KB，另外为了验证本设计的鲁棒性，丢包率增大到了10%。

本次实验的结果如下：



上图可知，可知本次实验结果符合预期，客户端发送的文件与服务器端接受的文件一致。

遇到的问题

本次实验中需要更进一步的理清TCP传输参数之间的关系。除了TCP重传部分设计比较耗时外，本次实验还存在一个问题（该问题上次实验应该就出现了）：

在设计tcp_app.c的server端时，由于接收端不清楚传输文件的总大小，因此无法准确的判断什么时候需要退出while循环。若不直接退出while循环，则不能正常进行fclose，这样会导致最后一段写入file的数据丢失。为了解决这一问题，本设计设计成若单词传输的大小<1460（单个数据包的最大数据长度），则退出循环。但这样的设计还是可能会导致一定的问题，若传输的总数据长度为1460的整数倍，则还是无法识别最后一个包的到来。

若要从根本上解决这一问题，应该是应用层的设计需要考虑的，例如一开始就传给对端接下来要传输数据的总长度（类似于HTTP协议的实现）。