- 网络传输机制实验（二）
  - 实验内容
    - 实验内容一
    - 实验内容二
  - 设计思路
    - 实现数据传输
      - `tcp_sock_read` 函数
      - `handle_recv_data` 函数
      - 更新后的`tcp_process` 函数
      - `tcp_sock_write` 函数
  - 结果验证
    - 实验一结果
    - 实验二结果

网络传输机制实验（二）

2020年12月30日

蔡润泽

本实验 `Github` 地址

实验内容

实验内容一

- 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，连接h1并正确收发数据 (./tcp_stack client 10.0.0.1 10001)
    - client向server发送数据，server将数据echo给client
- 使用tcp_stack.py替换其中任意一端，对端都能正确收发数据

实验内容二

- 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

实现数据传输

`tcp_sock_read` 函数

负责接收TCP数据，若ring buffer为空，则睡眠，当收到数据包时则被唤醒。具体实现如下：

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len) {
        while (ring_buffer_empty(tsk->rcv_buf)) {
                sleep_on(tsk->wait_recv);
        }
        int rlen = read_ring_buffer(tsk->rcv_buf, buf, len);
        wake_up(tsk->wait_recv);
        return rlen;
}
}
```

`handle_recv_data` 函数

该函数负责接收TCP数据包中的数据，根据ACK的值添加进ring buffer。另外，若ring buffer为满，则睡眠。具体实现如下：

```
void handle_recv_data(struct tcp_sock *tsk, struct tcp_cb *cb) {
        while (ring_buffer_full(tsk->rcv_buf)) {
                sleep_on(tsk->wait_recv);
        }
        write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
        wake_up(tsk->wait_recv);
        tsk->rcv_nxt = cb->seq + cb->pl_len;
        tsk->snd_una = cb->ack;
        tcp_send_control_packet(tsk, TCP_ACK);
}
```

更新后的`tcp_process`函数

相较于上周的实验，本周的实验需要对该函数进行补充，以支持接收TCP数据包。更新部分的代码如下：

```
if (tsk->state == TCP_ESTABLISHED) {
    if (tcp->flags & TCP_FIN) {
        tcp_set_state(tsk, TCP_CLOSE_WAIT);
        tsk->rcv_nxt = cb->seq + 1;
        tcp_send_control_packet(tsk, TCP_ACK);
    } else if (tcp->flags & TCP_ACK) {
        if (cb->pl_len == 0) {
            tsk->snd_una = cb->ack;
            tsk->rcv_nxt = cb->seq + 1;
            tcp_update_window_safe(tsk, cb);
        } else {
            handle_recv_data(tsk, cb);
        }
    }

}
```

`tcp_sock_write`函数

负责发送TCP数据包。如果对端recv_window允许，则发送数据，每次读取1个数据包大小的数据，即`min(data_len, 1514 - ETHER_HDR_SIZE - IP_HDR_SIZE - TCP_HDR_SIZE)`。然后封装数据包，通过IP层发送函数，将数据包发出去。具体实现如下：

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len) {
        int single_len = 0;
        int init_seq = tsk->snd_una;
        int init_len = len;

        while (len > 1514 - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BASE_HDR_SIZE) {
                single_len = min(len, 1514 - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE - TCP_BA
                send_data(tsk, buf + (tsk->snd_una - init_seq), single_len);
                sleep_on(tsk->wait_send);
                len -= single_len;
        }

        send_data(tsk, buf + (tsk->snd_una - init_seq), len);
        return init_len;
}
```

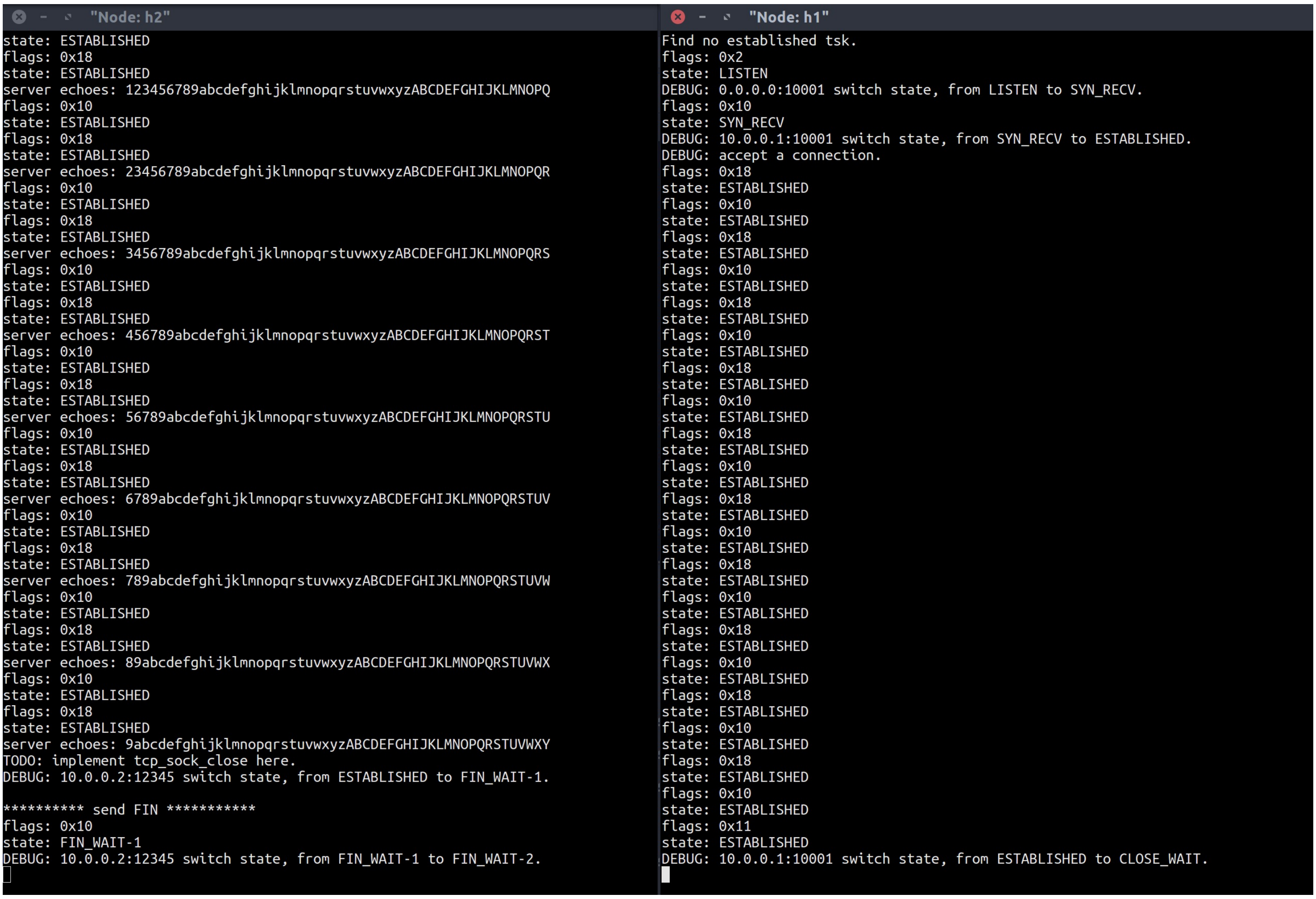
其中调用的`send_data`函数如下，负责调用tcp_send_packet函数发送数据包：

```
void send_data(struct tcp_sock *tsk, char *buf, int len) {
	int send_packet_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE + l
	char * packet = (char *)malloc(send_packet_len);
	memcpy(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE, buf, ler
	tsk->snd_wnd = len;
	tcp_send_packet(tsk, packet, send_packet_len);
}
```

结果验证

实验一结果

实验结果如下：

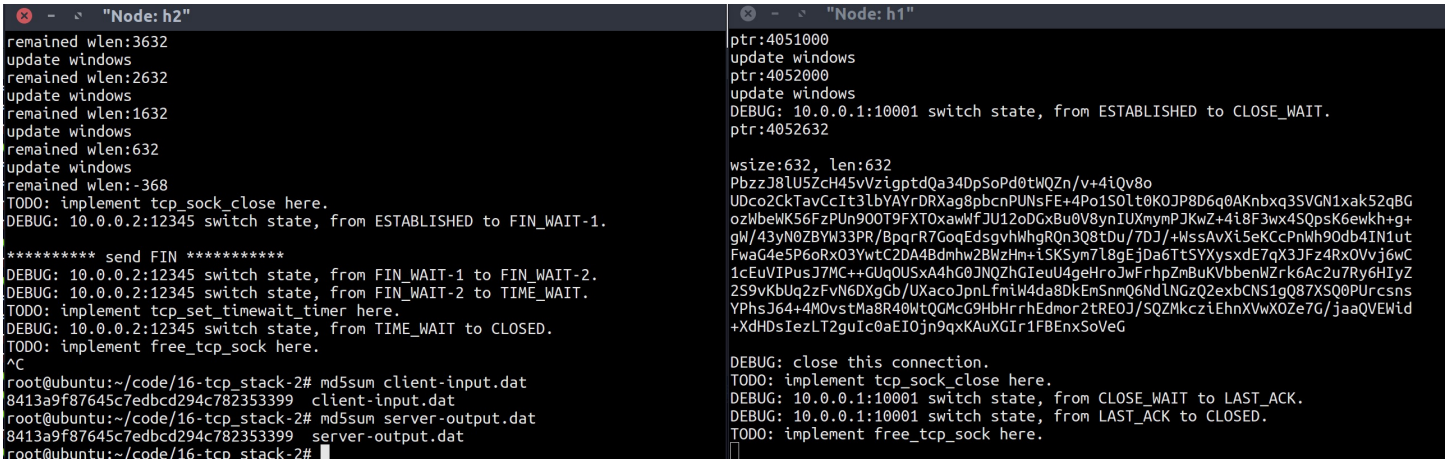


上图可知，可知本次实验结果符合预期。每次能echo正确的值。

另外，若h2用tcp_stack.py运行client，h1不管是运行本设计的server还是利用用tcp_stack.py运行server，结果都是echo出整个字符串“0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ”，结果也符合预期。

实验二结果

实验结果如下：



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。