网络传输机制实验1报告

张翔雨 2018K8009929035

一、实验题目：网络传输机制实验1

二、实验内容

- 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，连接至h1，显示建立连接成功后自动关闭连接 (./tcp_stack client 10.0.0.1 10001)
- 可以在一端用tcp_stack.py替换tcp_stack执行，测试另一端
- 通过wireshark抓包来来验证建立和关闭连接的正确性

三、实验过程

1.状态机转移

实现tcp_in.c中的void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)

主要功能是为了实现根据收到的包的不同类型和当前状态实现状态转移的过程。

```
void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
    //如果收到RST包则直接结束连接
    if(cb->flags & TCP_RST)
    {
        tcp_sock_close(tsk);
        free(tsk);
        return;
    }
```

```c
    //根据不同的状态收到不同的包类型进行状态转移
    switch(tsk->state)
    {
        case TCP_CLOSED:
            tcp_send_reset(cb);
            break;
        case TCP_LISTEN:
            if(cb->flags & TCP_SYN)
            {
                struct tcp_sock *child_sk = alloc_tcp_sock();
                child_sk->parent = tsk;
                child_sk->sk_sip = cb->daddr;
                child_sk->sk_sport = cb->dport;
                child_sk->sk_dip = cb->saddr;
                child_sk->sk_dport = cb->sport;
                child_sk->iss = tcp_new_iss();
                child_sk->rcv_nxt = cb->seq + 1;
                child_sk->snd_nxt = child_sk->iss;

                list_add_tail(&child_sk->list, &tsk->listen_queue);
                tcp_send_control_packet(child_sk, TCP_ACK | TCP_SYN);
                tcp_set_state(child_sk, TCP_SYN_RECV);
                tcp_hash(child_sk);
            }
            else
                tcp_send_reset(cb);
            break;
        case TCP_SYN_SENT:
            if(cb->flags & (TCP_SYN | TCP_ACK))
            {
                tsk->rcv_nxt = cb->seq + 1;
                tsk->snd_una = cb->ack;
                tcp_send_control_packet(tsk, TCP_ACK);
                tcp_set_state(tsk, TCP_ESTABLISHED);
                wake_up(tsk->wait_connect);
            }
            else
                tcp_send_reset(cb);
            break;
        case TCP_SYN_RECV:
            if(cb->flags & TCP_ACK)
            {
                tcp_sock_accept_enqueue(tsk);
                tsk->rcv_nxt = cb->seq;
                tsk->snd_una = cb->ack;
                wake_up(tsk->parent->wait_accept);
            }
            else
                tcp_send_reset(cb);
            break;
        case TCP_ESTABLISHED:
            if(cb->flags & TCP_FIN)
```

```
            {
                tsk->rcv_nxt = cb->seq+1;
                tcp_set_state(tsk, TCP_CLOSE_WAIT);
                tcp_send_control_packet(tsk, TCP_ACK);
            }
            break;
        case TCP_FIN_WAIT_1:
            if(cb->flags & TCP_ACK)
                tcp_set_state(tsk, TCP_FIN_WAIT_2);
            break;
        case TCP_FIN_WAIT_2:
            if(cb->flags & TCP_FIN)
            {
                tsk->rcv_nxt = cb->seq + 1;
                tcp_send_control_packet(tsk, TCP_ACK);
                tcp_set_state(tsk, TCP_TIME_WAIT);
                tcp_set_timewait_timer(tsk);
            }
            break;
        case TCP_LAST_ACK:
            if(cb->flags & TCP_ACK)
                tcp_set_state(tsk, TCP_CLOSED);
            break;
        default:
            break;
    }
}
```

2.TCP计时器管理

（1）实现tcp_scan_timer_list()

主要功能是扫描计时器列表，删除等待超过2*MSL的socket

```
void tcp_scan_timer_list()
{
    struct tcp_sock *tsk;
    struct tcp_timer *timer, *q;
    list_for_each_entry_safe(timer, q, &timer_list, list)
    {
        timer->timeout -= TCP_TIMER_SCAN_INTERVAL;
        if (timer->timeout <= 0)
        {
            list_delete_entry(&timer->list);
            tsk = timewait_to_tcp_sock(timer);
            if (!tsk->parent)
                tcp_bind_unhash(tsk);
            tcp_set_state(tsk, TCP_CLOSED);
            free_tcp_sock(tsk);
```

```
        }
    }
}
```

（2）实现`tcp_set_timewait_timer(struct tcp_sock *tsk)`

主要功能是设置socket计时器，并将其添加进计时器列表中。

```
void tcp_set_timewait_timer(struct tcp_sock *tsk)
{
    tsk->timewait.type = 0;
    tsk->timewait.timeout = TCP_TIMEWAIT_TIMEOUT;
    list_add_tail(&tsk->timewait.list, &timer_list);
    tsk->ref_cnt ++;
}
```

3.实现TCP Socket具体实现

（1）实现数据包信息查找对应的Socket

`struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport)`：在established_table中根据完整确定的四元组信息做查找。

```
struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport)
{
    int hash = tcp_hash_function(saddr, daddr, sport, dport);
    struct list_head * list = &tcp_established_sock_table[hash];

    struct tcp_sock *entry;
    list_for_each_entry(entry, list, hash_list) {
        if (saddr == entry->sk_sip   && daddr == entry->sk_dip
            && sport == entry->sk_sport && dport == entry->sk_dport)
            return entry;
    }
    return NULL;
}
```

`struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)`：在listen_table中根据单个key（sport）的信息做查找。

```
struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
{
    int hash = tcp_hash_function(0, 0, sport, 0);
    struct list_head *list = &tcp_listen_sock_table[hash];

    struct tcp_sock *entry;
    list_for_each_entry(entry, list, hash_list) {
        if (sport == entry->sk_sport)
            return entry;
    }
    return NULL;
}
```

(2)实现连接管理函数

`int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)`

主要功能：首先初始化四元组信息，然后将tcp socket 与bind_table做哈希，发送SYN包并转移到TCP_SYN_SENT状态，sleep on wait_connect来等待SYN包的到达

```
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
    // 1. initialize the four key tuple (sip, sport, dip, dport);
    int sport = tcp_get_port();
    if(tcp_sock_set_sport(tsk,sport) == -1)
        return -1;
    rt_entry_t* entry = longest_prefix_match(ntohl(skaddr->ip));
    tsk->sk_sip = entry->iface->ip;
    tsk->sk_dport = ntohs(skaddr->port);
    tsk->sk_dip = ntohl(skaddr->ip);
    // 2. hash the tcp sock into bind_table;
    tcp_bind_hash(tsk);
    // 3. send SYN packet, switch to TCP_SYN_SENT state, wait for the incoming
    //    SYN packet by sleep on wait_connect;
    tcp_send_control_packet(tsk, TCP_SYN);
    tcp_set_state(tsk, TCP_SYN_SENT);
    tcp_hash(tsk);
    sleep_on(tsk->wait_connect);
    return 0;
}
```

`int tcp_sock_listen(struct tcp_sock *tsk, int backlog)`

主要功能是设置backlog，将状态切换到listen状态，并进行hash操作添加到listen_table中。

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    tcp_hash(tsk);
    return 0;
}
```

`struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)`

主要功能是当accept队列不为空时释放第一个socket并且接受它否则sleep on wait_accept并等待接受

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
    while(list_empty(&tsk->accept_queue))
        sleep_on(tsk->wait_accept);

    struct tcp_sock *pop = tcp_sock_accept_dequeue(tsk);
    tcp_set_state(pop, TCP_ESTABLISHED);
    tcp_hash(pop);
    return pop;
}
```

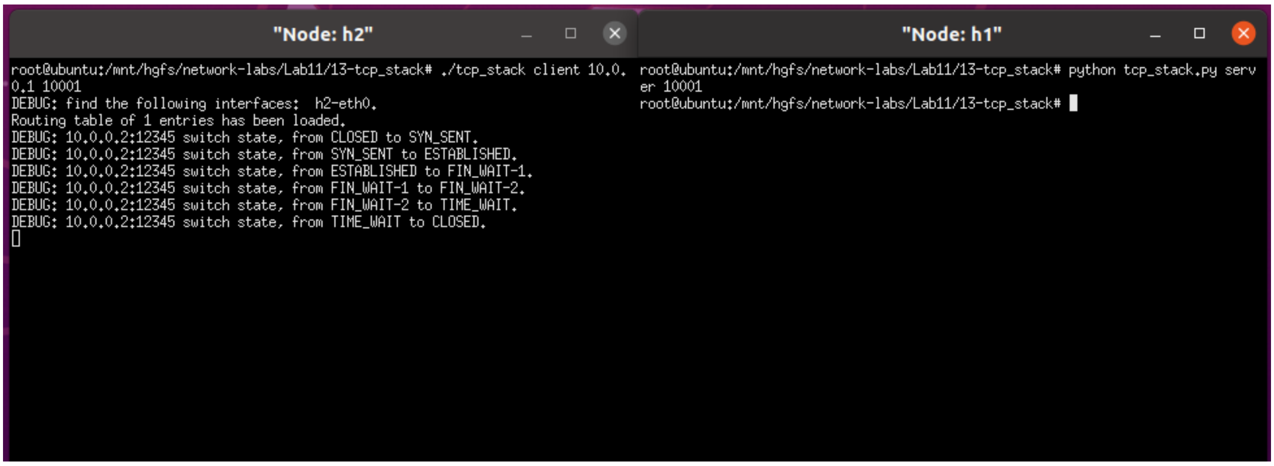
`void tcp_sock_close(struct tcp_sock *tsk)`

主要功能是根据不同的状态进行关闭连接时的发包处理并切换到对应的状态。

```
void tcp_sock_close(struct tcp_sock *tsk)
{
    switch (tsk->state) {
        case TCP_LISTEN:
            tcp_unhash(tsk);
            tcp_set_state(tsk, TCP_CLOSED);
            break;
        case TCP_ESTABLISHED:
            tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            break;
        case TCP_CLOSE_WAIT:
            tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
            tcp_set_state(tsk, TCP_LAST_ACK);
            break;
        default:
            break;
    }
}
```

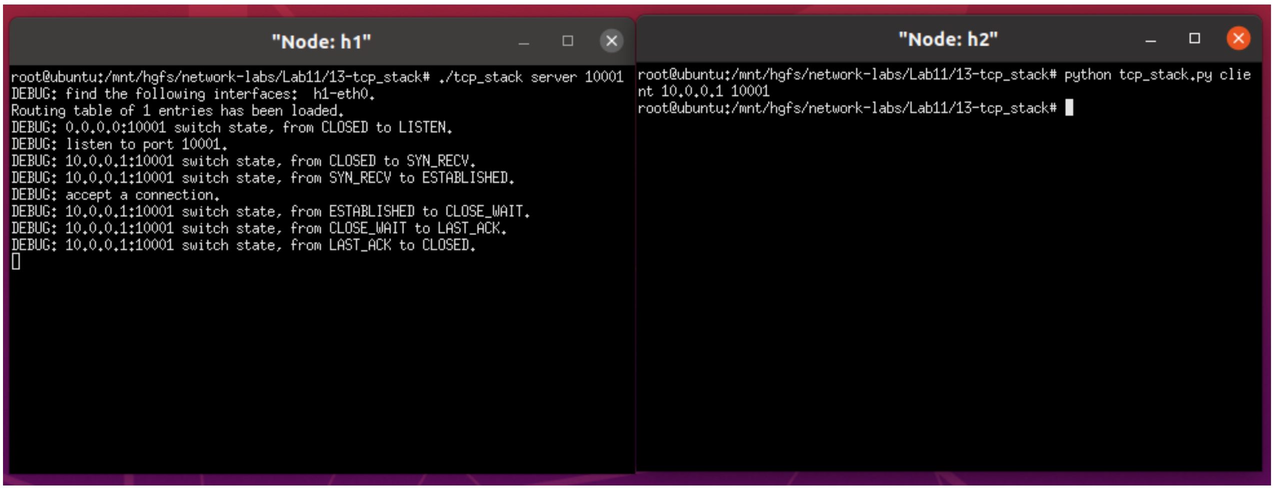
四、实验结果

1.server脚本与本次实验client交互结果



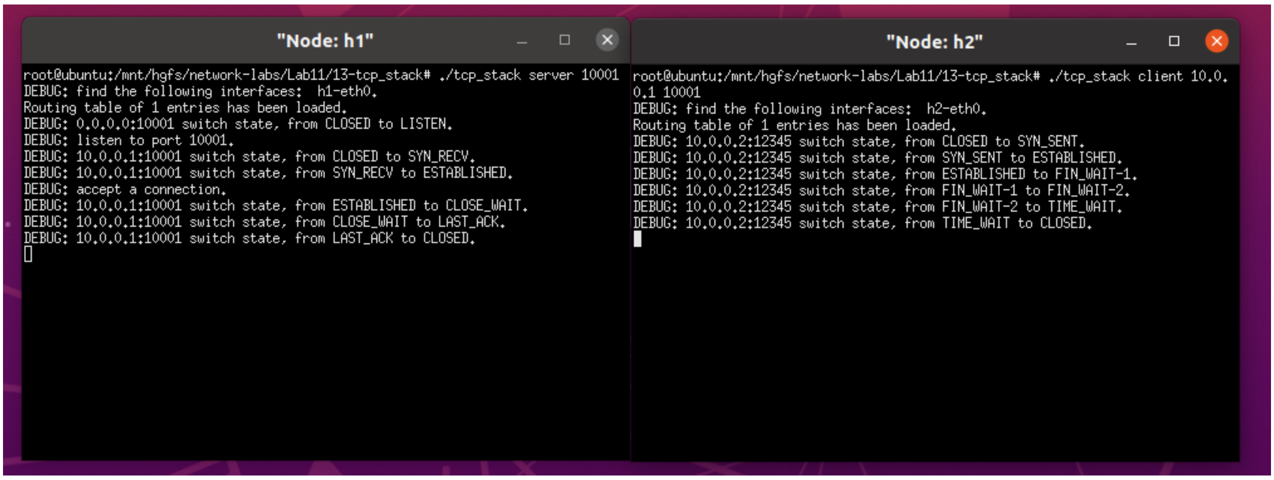
可以看到client正确地执行了状态转移：`CLOSED->SYN_SENT->ESTABLISHED->FIN_WAIT-1->FIN_WAIT_2->TIME_WAIT->CLOSED`，server脚本也正确退出。

2.client脚本与本次实验server交互结果



可以看到server正确执行了状态转移，子socket：`CLOSED->SYN_RECV->ESTABLISHED->CLOSE_WAIT->LAST_ACK->CLOSED`。client脚本也正确退出。

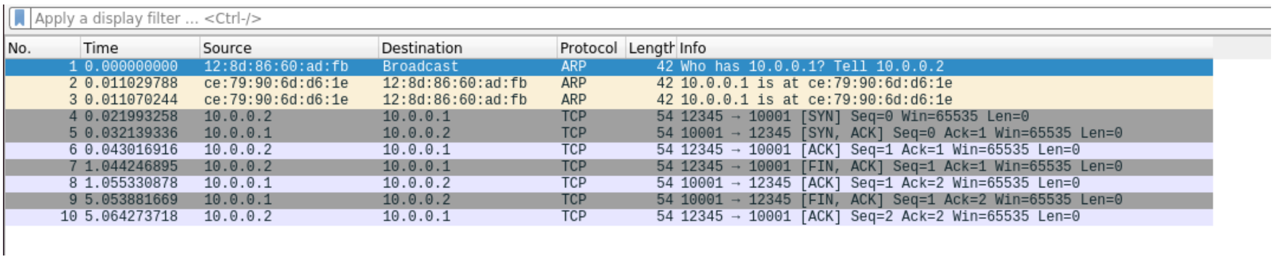
3.本次实验client与server交互



可以看到server正确执行了状态转移，子socket： `CLOSED->SYN_RECV->ESTABLISHED->CLOSE_WAIT->LAST_ACK->CLOSED` ，client正确地执行了状态转移： `CLOSED->SYN_SENT->ESTABLISHED->FIN_WAIT-1->FIN_WAIT_2->TIME_WAIT->CLOSED` 。

Wireshark抓包结果

本次实验实现的server与client交互结果抓包：



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	12:8d:86:60:ad:fb	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.011029788	ce:79:90:6d:d6:1e	12:8d:86:60:ad:fb	ARP	42	10.0.0.1 is at ce:79:90:6d:d6:1e
3	0.011070244	ce:79:90:6d:d6:1e	12:8d:86:60:ad:fb	ARP	42	10.0.0.1 is at ce:79:90:6d:d6:1e
4	0.021993258	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.032139336	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.043016916	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.044246895	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.055330878	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.053881669	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.064273718	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

可以看到整个过程如下：

建立连接时，经历三次握手：

1. h2向h1发送一个SYN包
2. h1收到SYN包，并发送SYN|ACK包
3. h2收到包，向h1发送ACK包

取消连接时，经历四次挥手：

1. h2向h1发送一个FIN|ACK包
2. h1收到包，向h2发送ACK包
3. h1收到包，向h2发送FIN|ACK包
4. h2收到包，回复ACK包

使用标准server与client脚本验证结果如下

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	a2:09:49:80:e4:ae	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010248908	5e:f8:fd:25:dd:96	a2:09:49:80:e4:ae	ARP	42	10.0.0.1 is at 5e:f8:fd:25:dd:96
3	0.021228278	10.0.0.2	10.0.0.1	TCP	74	55112 → 10001 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=…
4	0.031501687	10.0.0.1	10.0.0.2	TCP	74	10001 → 55112 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460…
5	0.042336540	10.0.0.2	10.0.0.1	TCP	66	55112 → 10001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=3993669…
6	1.043791789	10.0.0.2	10.0.0.1	TCP	66	55112 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=39…
7	1.055870361	10.0.0.1	10.0.0.2	TCP	66	10001 → 55112 [ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=1960505…
8	5.058177329	10.0.0.1	10.0.0.2	TCP	66	10001 → 55112 [FIN, ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=19…
9	5.068449792	10.0.0.2	10.0.0.1	TCP	66	55112 → 10001 [ACK] Seq=2 Ack=2 Win=42496 Len=0 TSval=3993674…

得到相同结果，实验成功。

六、实验总结

本次实验完全过渡到了新的协议层，但是难度不大，核心状态机的写法已经在verilog编程中练习过很多次，对照TCP建立连接的过程可以很好的构建。同时配合Wireshark的抓包结果更加直观的展现了TCP建立与取消连接的过程。