

计算机网络研讨课实验报告

冯吕 2015K8009929049

2018 年 6 月 22 日

1 实验题目

网络传输机制实验一

2 实验内容

在本次实验中，需要实现 *TCP* 的连接管理：建立连接和关闭连接。

在建立连接的过程中，需要完成三次握手。

3 实验流程

3.1 函数实现

在本次实验中，首先需要实现连接管理相关函数，需要实现的函数均位于 *tcp_sock.c* 和 *tcp_in.c* 两个文件中。

查找函数 *tcp_sock.c* 中首先需要实现两个查找函数，在 *established table* 和 *listen table* 中查找 *socket*。在 *established table* 中查找时，以四元组 *hash*，然后进行查找，而在 *listen table* 中进行查找时，只需使用 *sport*。

```
1 struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr,
2 u16 sport, u16 dport)
3 {
4     fprintf(stdout, "TODO: implement this function please.\n");
5
6     int key = tcp_hash_function(saddr, daddr, sport, dport);
7     struct tcp_sock *sk_pos, *sk_q;
8     list_for_each_entry_safe(sk_pos, sk_q, &
9 tcp_established_sock_table[key], hash_list){
10         if (saddr == sk_pos->local.ip && daddr == sk_pos->peer.ip \
11             && sport == sk_pos->local.port && dport == sk_pos->peer.port){
12             return sk_pos;
13         }
14     }
15     return NULL;
16 }
```

```

17
18 struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
19 {
20     fprintf(stdout, "TODO: implement this function please.\n");
21     u8 key = tcp_hash_function(0, 0, sport, 0);
22     struct tcp_sock *sk_pos, *sk_q;
23     list_for_each_entry_safe(sk_pos, sk_q, &tcp_listen_sock_table[key],
24     hash_list){
25         if (sk_pos->local.port == sport){
26             struct tcp_sock *child_pos, *child_q;
27             list_for_each_entry_safe(child_pos, child_q,
28             &sk_pos->listen_queue, list){
29                 if (child_pos->local.port == sport
30                 && child_pos->local.ip == saddr){
31                     return child_pos;
32                 }
33             }
34         }
35     }
36     return NULL;
37 }

```

tcp_sock_connect 函数 *tcp_sock_connect* 函数实现主动建立连接。主动建立连接时，四元组已经确定，发送 *SYN* 数据包，进入 *TCP_SYN_SENT* 状态，然后 *wait_connect*：

```

1 int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     tsk->peer.ip = ntohl(skaddr->ip);
5     tsk->peer.port = ntohs(skaddr->port);
6     tsk->local.port = tcp_get_port();
7     tsk->local.ip = 0x0a000002;
8
9     tcp_bind_hash(tsk);
10    tcp_send_control_packet(tsk, TCP_SYN);
11    tcp_set_state(tsk, TCP_SYN_SENT);
12    sleep_on(tsk->wait_connect);
13
14    tcp_hash(tsk);
15
16    return -1;
17 }

```

tcp_sock_listen 函数 该函数设置 *socket* 的 *backlog*, 然后进行 *TCP_LISTEN* 状态, 并将 *socket* hash 到 *listen table* 上:

```

1 int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     tsk->backlog = backlog;
5     tcp_set_state(tsk, TCP_LISTEN);
6     tcp_hash(tsk);
7     return 0;
8 }

```

tcp_sock_accept 函数 该函数实现被动连接的 *accept* 操作, 如果 *accept queue* 不为空, 则弹出一个 *socket* 来服务, 否则, *sleep*:

```

1 struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     if (!tsk->accept_backlog){
5         sleep_on(tsk->wait_accept);
6     }
7     if (tsk->accept_backlog){
8         struct tcp_sock *accept_tsk = tcp_sock_accept_dequeue(tsk);
9         tcp_hash(accept_tsk);
10        return accept_tsk;
11    }
12    return NULL;
13 }

```

tcp_state_listen 函数 该函数处理当处于 *TCP_LISTEN* 状态时收到包的操作: 首先判断是否为 *TCP_SYN* 包, 如果是, 则分配一个 *child socket* 来服务这个连接请求, 然后由 *child socket* 回复 *TCP_SYN* | *TCP_ACK* 包, 并将 *child socket* hash 到 *established table*。

```

1 void tcp_state_listen(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     if (cb->flags & TCP_SYN){
5         tsk->local.ip = cb->daddr;
6         tsk->local.port = cb->dport;
7         struct tcp_sock *child_sock = alloc_tcp_sock();
8         child_sock->local.ip = cb->daddr;
9         child_sock->local.port = cb->dport;
10        child_sock->peer.ip = cb->saddr;
11        child_sock->peer.port = cb->sport;
12        child_sock->parent = tsk;
13        tcp_set_state(child_sock, TCP_SYN_RECV);

```

```

14         list_add_head(&child_sock->list, &tsk->listen_queue);
15
16         tcp_send_control_packet(child_sock, TCP_SYN | TCP_ACK);
17         tcp_hash(child_sock);
18     }
19     else {
20         tcp_send_reset(cb);
21     }
22 }

```

tcp_state_syn_sent 函数 该函数处理主动建立连接的一方当收到 *TCP_SYN | TCP_ACK* 包时的处理：回复 *TCP_ACK* 包，然后进行 *TCP_ESTABLISHED* 状态，并唤醒 *wait_connect*。

```

1 void tcp_state_syn_sent(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     if (cb->flags & (TCP_SYN | TCP_ACK)){
5         tcp_send_control_packet(tsk, TCP_ACK);
6         tcp_set_state(tsk, TCP_ESTABLISHED);
7         wake_up(tsk->wait_connect);
8     }
9     else {
10         tcp_send_reset(cb);
11     }
12 }

```

tcp_state_syn_recv 函数 该函数处理当被动建立连接的一方收到主动建立连接的一方发送过来的 *ACK* 包时的处理过程：将自身从 *parent* 的 *listen queue* 移动到 *accept queue* 中，此时，连接已经建立起来了，进入 *TCP_ESTABLISHED* 状态。

```

1 void tcp_state_syn_recv(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     if (cb->flags & TCP_ACK){
5         tcp_sock_accept_enqueue(tsk);
6         tcp_set_state(tsk, TCP_ESTABLISHED);
7         tsk->peer.ip = cb->saddr;
8         tsk->peer.port = cb->sport;
9         wake_up(tsk->parent->wait_accept);
10    }
11    else {
12        tcp_send_reset(cb);
13    }
14 }

```

tcp_process 函数 该函数是处理一个收到包的完整流程，根据 *socket* 所处的状态和收到包的类型从而进行不同的处理：

```

1 void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
2 {
3     fprintf(stdout, "TODO: implement this function please.\n");
4     /*struct tcphdr *tcp_hdr = packet_to_tcp_hdr(packet);*/
5     /*struct iphdr *ip_hdr = packet_to_ip_hdr(packet);*/
6     if (tsk->state == TCP_CLOSED){
7         tcp_state_closed(tsk, cb, packet);
8     }
9
10    else if (tsk->state == TCP_LISTEN){
11        tcp_state_listen(tsk, cb, packet);
12    }
13
14    if (tsk->state == TCP_SYN_SENT){
15        tcp_state_syn_sent(tsk, cb, packet);
16    }
17
18    else{
19        if (!is_tcp_seq_valid(tsk, cb)){
20            free(packet);
21        }
22
23        else if (cb->flags & TCP_RST){
24            if (!tsk->parent){
25                tcp_bind_unhash(tsk);
26            }
27            tcp_set_state(tsk, TCP_CLOSED);
28            free_tcp_sock(tsk);
29        }
30
31        else if ((cb->flags & TCP_SYN) || (!(cb->flags & TCP_ACK))){
32            tcp_send_reset(cb);
33            if (!tsk->parent){
34                tcp_bind_unhash(tsk);
35            }
36            tcp_set_state(tsk, TCP_CLOSED);
37            free_tcp_sock(tsk);
38        }
39
40        else if (cb->flags & TCP_ACK){
41            if (cb->flags & TCP_FIN){
42                if (tsk->state == TCP_FIN_WAIT_1){

```

```

43         tcp_send_control_packet(tsk, TCP_ACK);
44         tcp_set_state(tsk, TCP_TIME_WAIT);
45         tcp_set_timewait_timer(tsk);
46     }
47     else {
48         tcp_set_state(tsk, TCP_CLOSE_WAIT);
49         tcp_sock_close(tsk);
50     }
51 }
52
53     else if (tsk->state == TCP_LAST_ACK){
54         if (!tsk->parent){
55             tcp_bind_unhash(tsk);
56         }
57         tcp_set_state(tsk, TCP_CLOSED);
58         free_tcp_sock(tsk);
59     }
60
61     else {
62         tcp_update_window_safe(tsk, cb);
63     }
64 }
65 }
66 }

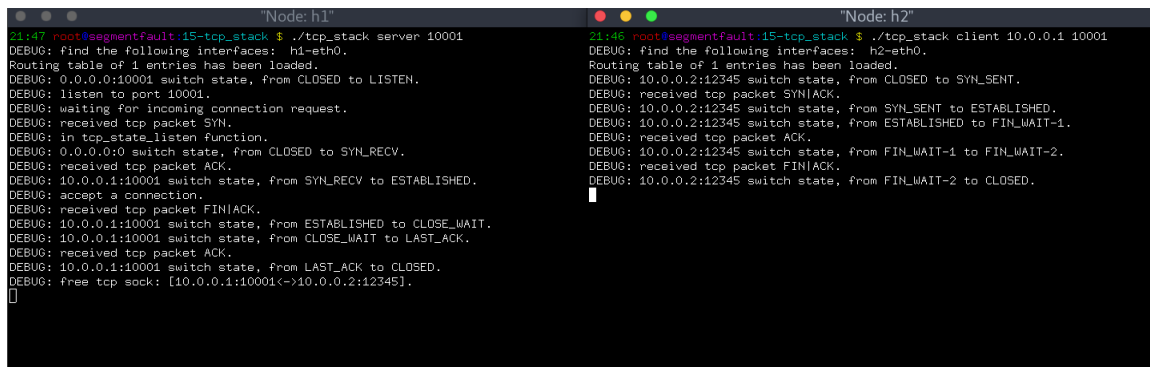
```

3.2 运行

- 运行给定网络拓扑：
 - 在节点 h_1 上执行 *TCP* 程序：执行脚本，禁止协议栈的相关功能，运行服务器模式；
 - 在节点 h_2 上执行 *TCP* 程序：执行脚本，禁止协议栈的相关功能，运行客户端模式，连接至 h_1 ，显示建立连接成功后自动关闭连接；
- 通过 *wireshark* 抓包来验证建立和关闭连接的正确性；

4 实验结果

运行 *TCP* 程序之后，能够成功建立连接：



```
"Node: h1"
21:47 root@segmentfault:15-tcp_stack $ ./tcp_stack server 10001
DEBUG: find the following interfaces: h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: waiting for incoming connection request.
DEBUG: received tcp packet SYN.
DEBUG: in tcp_state_listen function.
DEBUG: 0.0.0.0 switch state, from CLOSED to SYN_RECV.
DEBUG: received tcp packet ACK.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: received tcp packet FIN/ACK.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: received tcp packet ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: free tcp sock: [10.0.0.1:10001<->10.0.0.2:12345].
[]

"Node: h2"
21:46 root@segmentfault:15-tcp_stack $ ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces: h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: received tcp packet SYN/ACK.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: received tcp packet ACK.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: received tcp packet FIN/ACK.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to CLOSED.
```

5 结果分析

节点之间能够成功建立 *TCP* 连接并关闭，实验结果正确。