

- 网络传输机制实验（一）
 - 实验内容
 - 实验内容
 - 设计思路
 - TCP连接管理
 - tcp_sock_listen 函数
 - tcp_sock_accept 函数
 - tcp_sock_connect 函数
 - tcp_sock_close 函数
 - tcp_timer_thread 线程函数
 - 收包处理
 - tcp_sock_lookup_established 函数
 - tcp_sock_lookup_listen 函数
 - tcp_process 函数
 - 结果验证
 - 本设计client和标准server的输出结果
 - 本设计server和标准client的输出结果
 - 本设计server和本设计client的输出结果
 - wireshark抓包

网络传输机制实验（一）

2020年12月22日

蔡润泽

本实验 [Github](#) 地址

实验内容

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

设计思路

TCP连接管理

tcp_sock_listen 函数

若accept_queue非空，取出队列中第一个tcp sock 并accept, 否则, sleep on 互斥锁+信号wait_accept 。具体实现如下：

```
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_stack;
    if ((pop_stack = tcp_sock_accept_dequeue(tsk)) != NULL) {
        pop_stack->state = TCP_ESTABLISHED;
        tcp_hash(pop_stack);
        return pop_stack;
    } else {
        return NULL;
    }
}
```

tcp_sock_accept 函数

负责设置backlog，切换TCP_STATE到 TCP_LISTEN , 并且利用 hash_tcp 函数把TCP Sock加入到 listen_table 中。具体实现如下：

tcp_sock_connect 函数

1. 初始化四元组 (sip, sport, dip, dport);
2. 将tcp sock hash 到 into bind_table ;
3. 发送 SYN 信号包, 切换TCP状态到 TCP_SYN_SENT, 通过sleep on wait_connect 来等待 SYN包;
4. 若SYN 到达, 该线程被唤醒, 说明此时连接已经建立完成。具体实现如下：

```

int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr) {
    u16 sport = tcp_get_port();
    if (sport == 0) {
        return -1;
    }
    u32 saddr = longest_prefix_match(ntohl(skaddr->ip))->iface->ip;
    tsk->sk_sip = saddr;
    tsk->sk_sport = sport;
    tsk->sk_dip = ntohl(skaddr->ip);
    tsk->sk_dport = ntohs(skaddr->port);
    tcp_bind_hash(tsk);
    tcp_send_control_packet(tsk, TCP_SYN);
    tcp_set_state(tsk, TCP_SYN_SENT);
    tcp_hash(tsk);
    sleep_on(tsk->wait_connect);
    return sport;
}

```

tcp_sock_close 函数

该函数负责根据不同的TCP状态进行关闭连接，并切换到相应的状态。具体实现如下：

```

void tcp_sock_close(struct tcp_sock *tsk) {
    if (tsk->state == TCP_CLOSE_WAIT) {
        tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
        tcp_set_state(tsk, TCP_LAST_ACK);
    } else if (tsk->state == TCP_ESTABLISHED) {
        tcp_set_state(tsk, TCP_FIN_WAIT_1);
        tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
    } else {
        tcp_unhash(tsk);
        tcp_set_state(tsk, TCP_CLOSED);
    }
}

```

tcp_timer_thread 线程函数

该函数为定时器线程，定期扫描。对超过等待 $2 * \text{MSL}$ 时间，进入TCP_CLOSED状态，结束TCP_TIME_WAIT状态的流。

收包处理

节点在收到一个TCP数据包后，会首先进行TCP查找，找到连接相应的Sock并返回，之后再根据包的种类的TCP Sock对应的状态进行数据包的处理。

tcp_sock_lookup_established 函数

在进行TCP Sock查找时首先会根据四元组查找已经建立好的的Sock，若找到则返回对应的Sock。这部分涉及到的代码如下：

```

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

```

tcp_sock_lookup_listen 函数

若TCP Sock查找时没有找到已经完成建立的Sock，则会进行对处在Listen状态下的Sock进行查找，若找到则返回对应的Sock。这部分涉及到的代码如下：

```

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 *tmp;
    list_for_each_entry(tmp, list, hash_list) {
        if (sport == tmp->sk_sport) {
            return tmp;
        }
    }
    return NULL;
}

```

tcp_process 函数

该函数负责根据TCP状态机的不同状态进行数据包处理已经状态切换。具体实现如下：

```

void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet) {
    struct tcphdr * tcp = packet_to_tcp_hdr(packet);
    if ((tcp->flags & TCP_RST) == TCP_RST) {
        tcp_sock_close(tsk);
        return;
    }
    if (tsk->state == TCP_LISTEN) {
        if ((tcp->flags & TCP_SYN) == TCP_SYN) {
            tcp_set_state(tsk, TCP_SYN_RECV);
            struct tcp_sock * child = alloc_child_tcp_sock(tsk, cb);
            tcp_send_control_packet(child, TCP_ACK|TCP_SYN);
        }
    }
    if (tsk->state == TCP_SYN_SENT) {
        if ((tcp->flags & TCP_ACK) == TCP_ACK) {
            wake_up(tsk->wait_connect);
            tsk->rcv_nxt = cb->seq + 1;
            tsk->snd_una = cb->ack;
            tcp_send_control_packet(tsk, TCP_ACK);
            tcp_set_state(tsk, TCP_ESTABLISHED);
        }
    }
    if (tsk->state == TCP_SYN_RECV) {
        if ((tcp->flags & TCP_ACK) == TCP_ACK) {
            if (!tcp_sock_accept_queue_full(tsk)) {
                struct tcp_sock * csk = tcp_sock_listen_dequeue(tsk);
                tcp_sock_accept_enqueue(csk);
                if (!is_tcp_seq_valid(csk, cb)) {
                    return;
                }
                csk->rcv_nxt = cb->seq;
                csk->snd_una = cb->ack;
                tcp_set_state(csk, TCP_ESTABLISHED);
                wake_up(tsk->wait_accept);
            }
        }
    }
    if (!is_tcp_seq_valid(tsk, cb)) {
        return;
    }
    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);
        }
    }
    if (tsk->state == TCP_LAST_ACK) {
        if ((tcp->flags & TCP_ACK) == TCP_ACK) {
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
        }
    }
    if (tsk->state == TCP_FIN_WAIT_1) {

```

```

        if ((tcp->flags & TCP_ACK) == TCP_ACK) {
            tcp_set_state(tsk, TCP_FIN_WAIT_2);
        }
    }
    if (tsk->state == TCP_FIN_WAIT_2) {
        if ((tcp->flags & TCP_FIN) == 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);
        }
    }
}

```

其中，`alloc_child_tcp_sock`函数负责通过建立一个child sock。

结果验证

本设计client和标准server的输出结果

实验结果如下：

Node: h1	Node: h2
<pre> root@ubuntu:~/code/15-tcp_stack# python tcp_stack.py server 10001 root@ubuntu:~/code/15-tcp_stack# </pre>	<pre> rcv_nxt:0, ack:1, seq:4276322648 DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED. TODO: implement tcp_sock_close here. DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1. ***** send FIN ***** TODO: implement tcp_sock_lookup_established here. TODO: implement tcp_process here. flags: 0x10 state: FIN_WAIT-1 rcv_nxt:4276322649, ack:2, seq:4276322649 DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2. TODO: implement tcp_sock_lookup_established here. TODO: implement tcp_process here. flags: 0x11 state: FIN_WAIT-2 rcv_nxt:4276322649, ack:2, seq:4276322649 DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT. TODO: implement tcp_set_timewait_timer here. DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED. </pre>

上图可知，可知本次实验结果符合预期。

本设计server和标准client的输出结果

实验结果如下：

"Node: h1"

flags: 0x10
state: SYN_RECV
rcv_nxt:0, ack:846930887, seq:1401151680
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
ERROR: received packet with invalid seq, drop it.
DEBUG: accept a connection.
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x11
state: ESTABLISHED
rcv_nxt:1401151680, ack:846930887, seq:1401151680
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
TODO: implement tcp_sock_close here.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x10
state: LAST_ACK
rcv_nxt:1401151681, ack:846930888, seq:1401151681
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.

"Node: h2"

root@ubuntu:~/code/15-tcp_stack# python tcp_stack.py client 10.0.0.1 10001
root@ubuntu:~/code/15-tcp_stack#

上图可知，可知本次实验结果符合预期。

本设计server和本设计client的输出结果

实验结果如下：

"Node: h1"

TODO: implement tcp_process here.
flags: 0x10
state: SYN_RECV
rcv_nxt:0, ack:846930887, seq:1
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x11
state: ESTABLISHED
rcv_nxt:1, ack:846930887, seq:1
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
TODO: implement tcp_sock_close here.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x10
state: LAST_ACK
rcv_nxt:2, ack:846930888, seq:2
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.

"Node: h2"

rcv_nxt:0, ack:1, seq:846930886
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
TODO: implement tcp_sock_close here.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.

***** send FIN *****
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x10
state: FIN_WAIT-1
rcv_nxt:846930887, ack:2, seq:846930887
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
TODO: implement tcp_sock_lookup_established here.

TODO: implement tcp_process here.
flags: 0x11
state: FIN_WAIT-2
rcv_nxt:846930887, ack:2, seq:846930887
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
TODO: implement tcp_set_timewait_timer here.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.

上图可知，可知本次实验结果符合预期。

wireshark抓包

wireshark抓包结果如下：

Capturing from h1-eth0						
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help						
Apply a display filter ... <Ctrl-/> Expression...						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	82:9d:87:22:89:ca	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010113221	8a:a1:27:79:a0:88	82:9d:87:22:89:ca	ARP	42	10.0.0.1 is at 8a:a1:27:79:a0:88
3	0.010310931	8a:a1:27:79:a0:88	82:9d:87:22:89:ca	ARP	42	10.0.0.1 is at 8a:a1:27:79:a0:88
4	0.020618832	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.031254874	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.041401003	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.042852524	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.053622786	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.052811986	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.062895330	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0
11	332.653105815	82:9d:87:22:89:ca	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
12	332.664095577	8a:a1:27:79:a0:88	82:9d:87:22:89:ca	ARP	42	10.0.0.1 is at 8a:a1:27:79:a0:88
13	332.675360212	10.0.0.2	10.0.0.1	TCP	74	56160 → 10001 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1
14	332.686676520	10.0.0.1	10.0.0.2	TCP	74	10001 → 56160 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 S
15	332.697410637	10.0.0.2	10.0.0.1	TCP	66	56160 → 10001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=4285799919
16	333.698758930	10.0.0.2	10.0.0.1	TCP	66	56160 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=4285
17	333.710642588	10.0.0.1	10.0.0.2	TCP	66	10001 → 56160 [ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=333291618
18	337.713673775	10.0.0.1	10.0.0.2	TCP	66	10001 → 56160 [FIN, ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=3332
19	337.724308502	10.0.0.2	10.0.0.1	TCP	66	56160 → 10001 [ACK] Seq=2 Ack=2 Win=42496 Len=0 TSval=4285804948

Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0

Ethernet II, Src: 82:9d:87:22:89:ca (82:9d:87:22:89:ca), Dst: Broadcast (ff:ff:ff:ff:ff:ff)

Address Resolution Protocol (request)

h1-eth0: <live capture in progress>

Packets: 19 · Displayed: 19 (100.0%)

Profile: Default

上图中，蓝色部分为本实验中Server和Client的结果，红色部分为标准Server和Client的结果。通过对比可知本次实验结果符合预期。